Attorney Docket No.: 9099-4 PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re: Robert D. Black
Serial No.: 10/005,889
Filed: November 7, 2001
For: CIRCUITS FOR IN VIVO DETECTION OF BIOMOLECULE CONCENTRATIONS USING FLUORESCENT TAGS

Group: 1641
Examiner: Gary W. Counts
Confirmation No.: 7939

February 22, 2007

Mail Stop Appeal-Brief Patents
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

APPELLANT'S REPLY BRIEF ON APPEAL UNDER 37 C.F.R. §41.41

Sir:

This Reply Brief is filed in response to the Examiner's Answer mailed Janury 29, 2007.

It is not believed that an extension of time and/or additional fee(s) are required, beyond those that may otherwise be provided for in documents accompanying this paper. In the event, however, that an extension of time is necessary to allow consideration of this paper, such an extension is hereby petitioned for under 37 C.F.R. §1.136(a). Any additional fees believed to be due in connection with this paper may be charged to Deposit Account No. 50-0220.

I. The Examiner's Answer – Response to Arguments (starting at Page 10)

In the interest of brevity, Appellant will refrain herein from readdressing all of the deficiencies with the pending rejections and, therefore, hereby incorporates the arguments set out in Appellant's Brief on Appeal as if set forth in their entirety. Appellant's following remarks address the new points raised in the Examiner's Answer.

Regarding the Examiner's response to Appellant's remarks on the recitations of "configured to" in Appellant's claims, Appellant respectfully reiterates that recitations, such as "configured to" that require steps be performed do limit claims to a particular structure. In particular, the present claims clearly recite a process to be carried out by the processor circuit (such as the release of antibodies, waiting a predetermined time period, then activating radiation source, then waiting a second predetermined time interval, then sensing the voltages generated by radiation detector, etc.), which does impart a structure to the processor circuit.

Appellant believes that the Examiner's citation to MPEP § 2106 indicates that the Examiner considers the recitation of "configured to" to be equivalent to language such as "whereby" or "adapted to."[1] However, Appellant respectfully points out that the MPEP § 2106 also refers the reader to MPEP § 2111.04,which reads in-part:

> The determination of whether each of these clauses is a limitation in a claim depends on the specific facts of the case. In *Hoffer v. Microsoft Corp.*, 405 F.3d 1326, 1329, 74 USPQ2d 1481, 1483 (Fed. Cir. 2005), the court held that when a "'whereby' clause states a condition that is material to patentability, it cannot be ignored in order to change the substance of the invention." *Id*

In holding that the whereby clause did impart patentable weight to the claims, the court in Hoffer held that the capability of the system "is more than the intended result of a process step; it is part of the process itself. This interactive element is described in the specification and prosecution history as an integral part of the invention." *Id.* Therefore, Appellant respectfully maintains that the term "configured to" does means that the processor circuit does, in-fact, perform the recited process and, therefore, are not optional.

With respect to the Examiner's remarks regarding language such as "algorithm", Appellant respectfully maintains that the absence of the word "algorithm," by itself, is not conclusive evidence that no step by step process is disclosed. See *AT&T Corp. v. Excel Communications, Inc.*, 172 F.3d 1352, 1356 (Fed. Cir. 1999) ("This court recently pointed out that *any step-by-step process, be it electrical, chemical or mechanical*, involves an "algorithm" in the broad sense of the term" (emphasis added)). Accordingly, Appellant respectfully maintains that the processor circuits, as recited in the present claims, are structurally different than the prior art despite the absence of the terminology suggested by the Examiner.

---

[1] Further, Appellant believes that the Examiner may actually be mis-understanding the term "configured to" to mean "configurable" based on the assertion that this recitation only "suggests or makes optional but does not require steps to be performed." *Examiner's Answer, page 11.*

With regard to the Examiner's remarks on the term "programmed" not being recited in the claims, Appellant respectfully points out that previously issued patents indicate that it is not necessary to include the specific word "programmed" in a processor claim when disclosed in the function of a processor circuit. For example, see U.S. Patent Nos. 6,496,187, 6,457,117, and 6,876,704. (copies of which are attached hereto). Accordingly, Appellant respectfully maintains that the absence of the word "programmed" from the claims is not convincing evidence that the processor circuit claims are not structurally different.

II. Conclusion

For the reasons set forth in above and in Appellant's Brief on Appeal, Appellant requests reversal of the rejections, allowance of all claims and passing of the application to issue.

Respectfully submitted,

[illegible]

Robert N. Crouse
Registration No. 44,635

Myers Bigel Sibley & Sajovec, P.A.
P. O. Box 37428
Raleigh, North Carolina 27627
Telephone: (919) 854-1400
Facsimile: (919) 854-1401
Customer Number 20792

CERTIFICATION OF TRANSMISSION

I hereby certify that this correspondence is being transmitted via the Office electronic filing system in accordance with § 1.6(a)(4) to the U.S. Patent and Trademark Office on February 22, 2007.

Signature: [illegible]
Sheena Donnelly

US006457117B1

(12) United States Patent
Witt

(10) Patent No.: US 6,457,117 B1
(45) Date of Patent: Sep. 24, 2002

(54) PROCESSOR CONFIGURED TO PREDECODE RELATIVE CONTROL TRANSFER INSTRUCTIONS AND REPLACE DISPLACEMENTS THEREIN WITH A TARGET ADDRESS

(75) Inventor: **David B. Witt**, Austin, TX (US)

(73) Assignee: **Advanced Micro Devices, Inc.**, Sunnyvale, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/708,216**

(22) Filed: **Nov. 7, 2000**

Related U.S. Application Data

(63) Continuation of application No. 09/065,681, filed on Apr. 23, 1998, now Pat. No. 6,167,506.
(60) Provisional application No. 60/065,878, filed on Nov. 17, 1997.

(51) **Int. Cl.**^{7} **G06F 9/312**
(52) **U.S. Cl.** **712/213**; 712/204; 712/206; 712/207; 712/211; 712/213; 712/227; 712/235; 712/237; 712/239
(58) **Field of Search** 712/213, 204, 712/206, 207, 211, 227, 235, 237, 239

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,502,111 A	2/1985	Riffe et al.
5,101,341 A	3/1992	Circello et al.

(List continued on next page.)

FOREIGN PATENT DOCUMENTS

EP	0 238 810 A2	9/1987
EP	0 423 726 A2	4/1991

(List continued on next page.)

OTHER PUBLICATIONS

Wallace et al., "Multiple Branch and Block Prediction," ©1997 IEEE, pp. 94–103.
Tamasulo, "An Efficient Algorithm For Exploiting Multiple Arithmetic Units," 1967, IBM Journal, pp. 25–33.

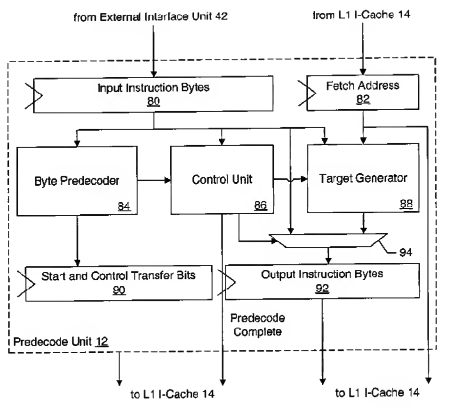
(List continued on next page.)

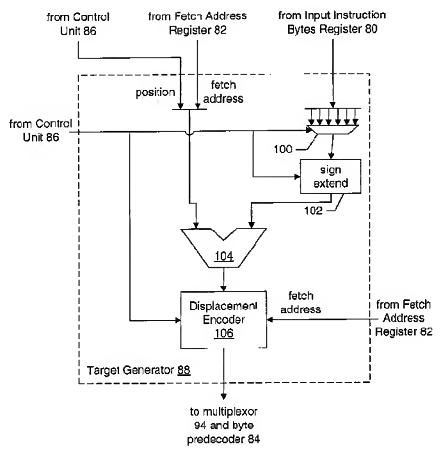
Primary Examiner—Richard L. Ellis
Assistant Examiner—Gautam R. Patel
(74) *Attorney, Agent, or Firm*—Conley, Rose & Tayon, PC; Lawrence J. Merkel

(57) ABSTRACT

The processor is configured to predecode instruction bytes prior to their storage within an instruction cache. During the predecoding, relative branch instructions are detected. The displacement included within the relative branch instruction is added to the address corresponding to the relative branch instruction, thereby generating the target address. The processor replaces the displacement field of the relative branch instruction with an encoding of the target address, and stores the modified relative branch instruction in the instruction cache. The branch prediction mechanism may select the target address from the displacement field of the relative branch instruction instead of performing an addition to generate the target address. In one embodiment, relative branch instructions having eight bit and 32-bit displacement fields are included in the instruction set executed by the processor. Additionally, the processor employs predecode information (stored in the instruction cache with the corresponding instruction bytes) including a start bit and a control transfer bit corresponding to each instruction byte. The combination of the start bit indicating that the byte is the start of an instruction and the corresponding control transfer bit identifies the instruction as either a branch instruction or a non-branch instruction. For relative branch instructions including an eight bit displacement, the control transfer bit corresponding to the displacement field is used in conjunction with the displacement field to store the encoded target address. Thirty-two bit displacement fields store the entirety of the target address, and hence the encoded target address comprises the target address.

14 Claims, 15 Drawing Sheets




from Control Unit 86
from Fetch Address Register 82
from Input Instruction Bytes Register 80
position
fetch address
from Control Unit 86
100
sign extend
102
104
Displacement Encoder 106
fetch address
from Fetch Address Register 82
Target Generator 88
to multiplexor 94 and byte predecoder 84

U.S. PATENT DOCUMENTS

5,129,067	A		7/1992	Johnson
5,155,820	A		10/1992	Gibson
5,233,696	A		8/1993	Suzuki
5,313,605	A		5/1994	Huck et al.
5,337,415	A		8/1994	DeLano et al.
5,438,668	A		8/1995	Coon et al.
5,459,844	A		10/1995	Eickemeyer et al.
5,488,710	A		1/1996	Sato et al.
5,499,204	A		3/1996	Barrera et al.
5,513,330	A		4/1996	Stiles
5,557,271	A		9/1996	Rim et al.
5,559,975	A		9/1996	Christie et al.
5,560,028	A		9/1996	Sachs et al.
5,566,298	A		10/1996	Boggs et al.
5,586,276	A		12/1996	Growchowski et al.
5,586,277	A		12/1996	Brown
5,598,544	A		1/1997	Oshima
5,600,806	A		2/1997	Brown et al.
5,608,886	A		3/1997	Blomgren et al.
5,625,787	A		4/1997	Mahin et al.
5,630,082	A		5/1997	Yao et al.
5,644,744	A		7/1997	Mahin et al.
5,689,672	A		11/1997	Witt et al.
5,692,168	A		11/1997	McMahan
5,729,707	A		3/1998	Maki
5,737,576	A	*	4/1998	Breternitz, Jr. 711/169
5,758,114	A		5/1998	Johnson et al.
5,758,116	A		5/1998	Lee et al.
5,819,059	A		10/1998	Tran
5,822,558	A		10/1998	Tran
5,860,152	A		1/1999	Savakar
5,872,943	A		2/1999	Pickett et al.
5,935,238	A		8/1999	Talcott et al.
5,968,163	A		10/1999	Narayan et al.
5,987,235	A		11/1999	Tran
6,049,863	A		4/2000	Tran et al.
6,061,786	A		5/2000	Witt
6,134,649	A		10/2000	Witt

FOREIGN PATENT DOCUMENTS

EP	0 498 654 A2	12/1992
EP	0 651 322 A1	5/1995
EP	0 651 324	5/1995
EP	0 718 758	6/1996
GB	2 263 987 A	8/1993
WO	96/10783	4/1996

OTHER PUBLICATIONS

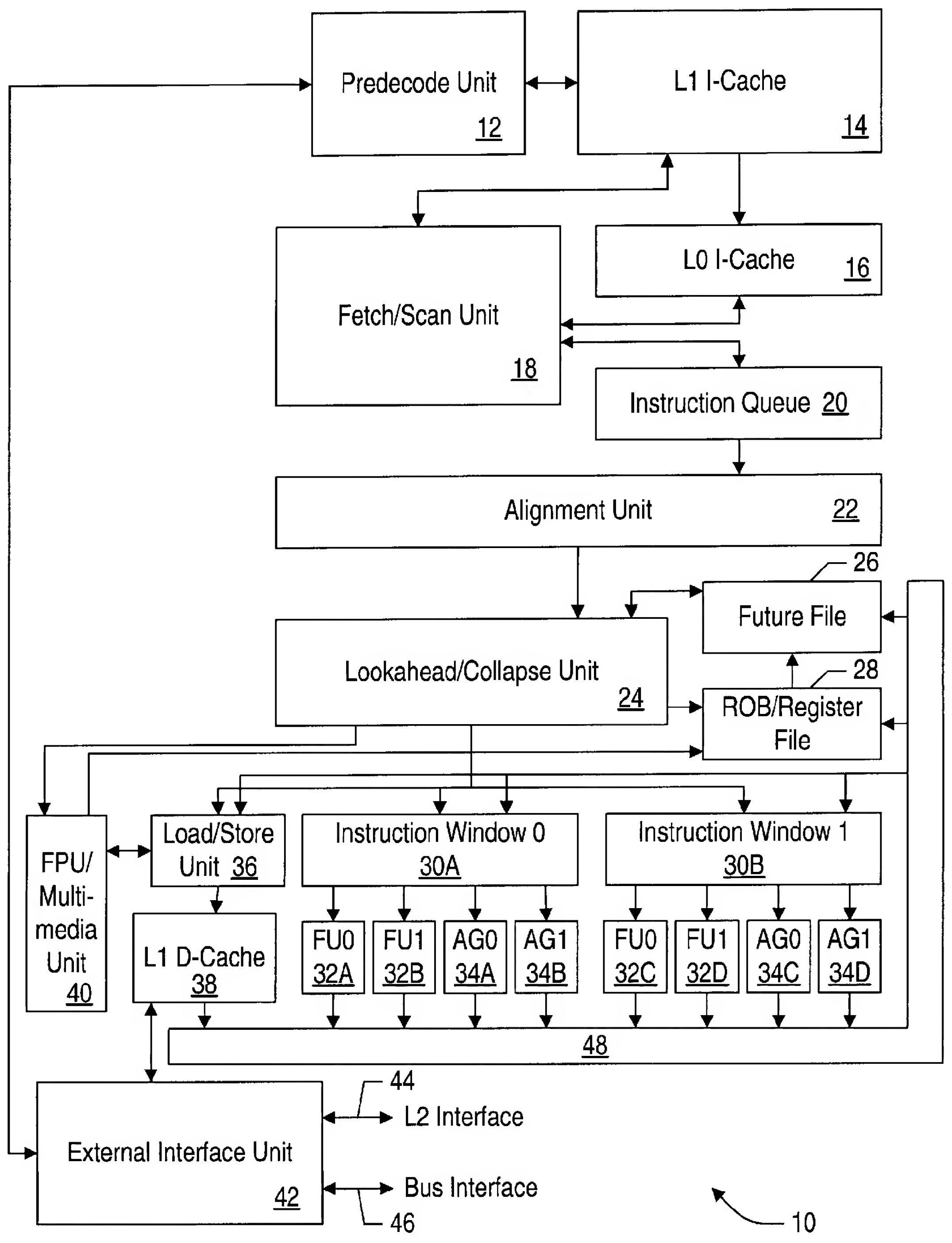
Minagawa, "Predecoding Mechanism for Superscaler Architecture," 1991, IEEE Publication, pp. 21–24.

XP000212073 Ditzel, et al., "Branch Folding in the CRISP Microprocessor: Reducing Branch Delay to Zero," AT&T Bell Laboratories, 14th Annual International Symposium on Computer Architecture, Jun. 2–5, 1987, pp. 2–9.

XP000364329 Gonzalez, et al., "Reducing Branch Delay to Zero in Pipelined Processors," IEEE Transactions on Computers, vol. 42, No. 3, Mar. 1993, pp. 363–371.

International Search Report for PCT/US 98/19045 mailed Dec. 28, 1998.

* cited by examiner



Predecode Unit 12
L1 I-Cache 14
Fetch/Scan Unit 18
L0 I-Cache 16
Instruction Queue 20
Alignment Unit 22
26
Future File
Lookahead/Collapse Unit 24
28
ROB/Register File
FPU/ Multi-media Unit 40
Load/Store Unit 36
Instruction Window 0 30A
Instruction Window 1 30B
L1 D-Cache 38
FU0 32A
FU1 32B
AG0 34A
AG1 34B
FU0 32C
FU1 32D
AG0 34C
AG1 34D
48
44
L2 Interface
External Interface Unit 42
Bus Interface
46
10

FIG. 1

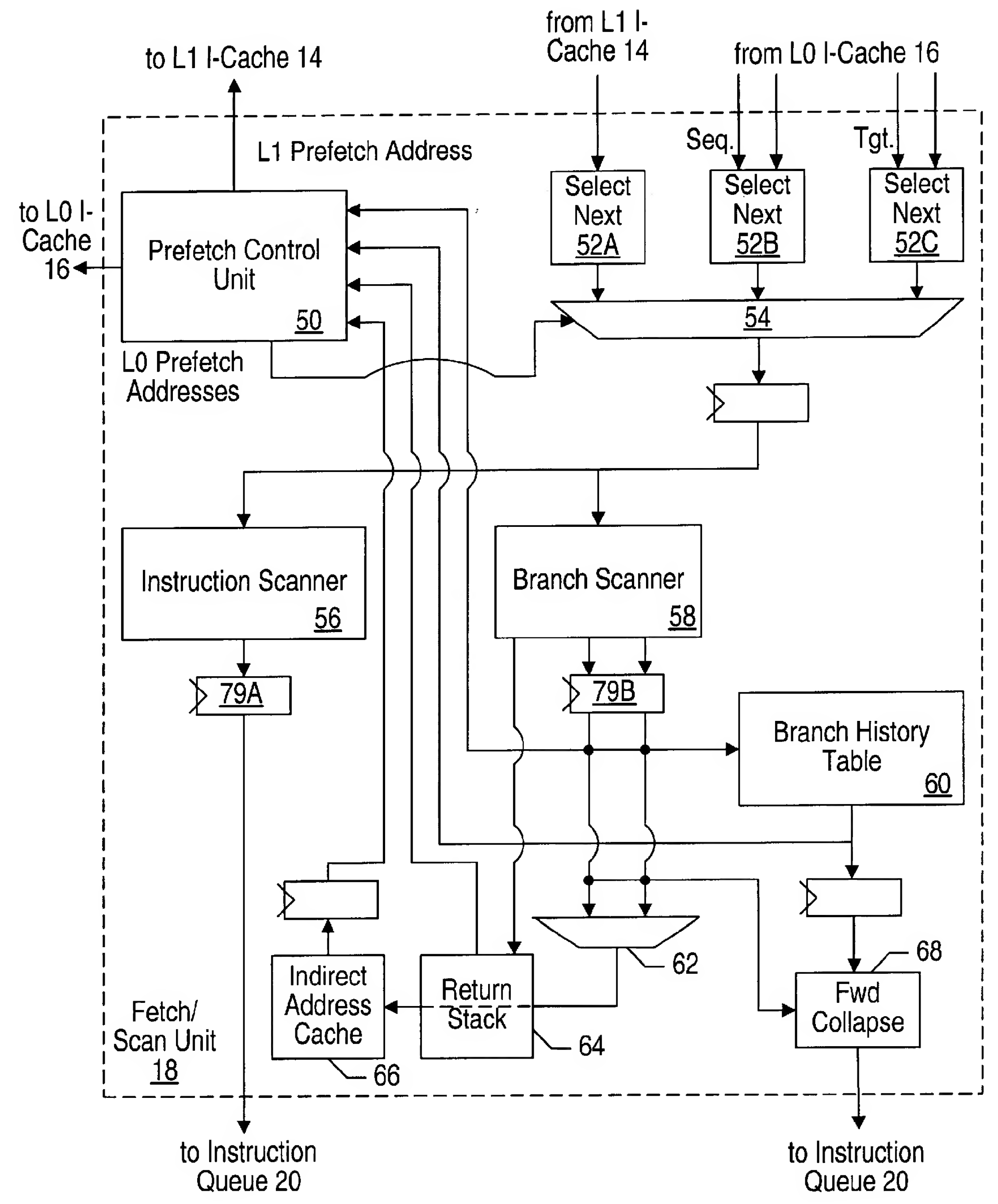

to L1 I-Cache 14
from L1 I-Cache 14
from L0 I-Cache 16
L1 Prefetch Address
Seq.
Tgt.
Select Next 52A
Select Next 52B
Select Next 52C
to L0 I-Cache 16
Prefetch Control Unit 50
54
L0 Prefetch Addresses
Instruction Scanner 56
Branch Scanner 58
79A
79B
Branch History Table 60
62
68
Indirect Address Cache
Return Stack
Fwd Collapse
64
66
Fetch/ Scan Unit 18
to Instruction Queue 20
to Instruction Queue 20

FIG. 2

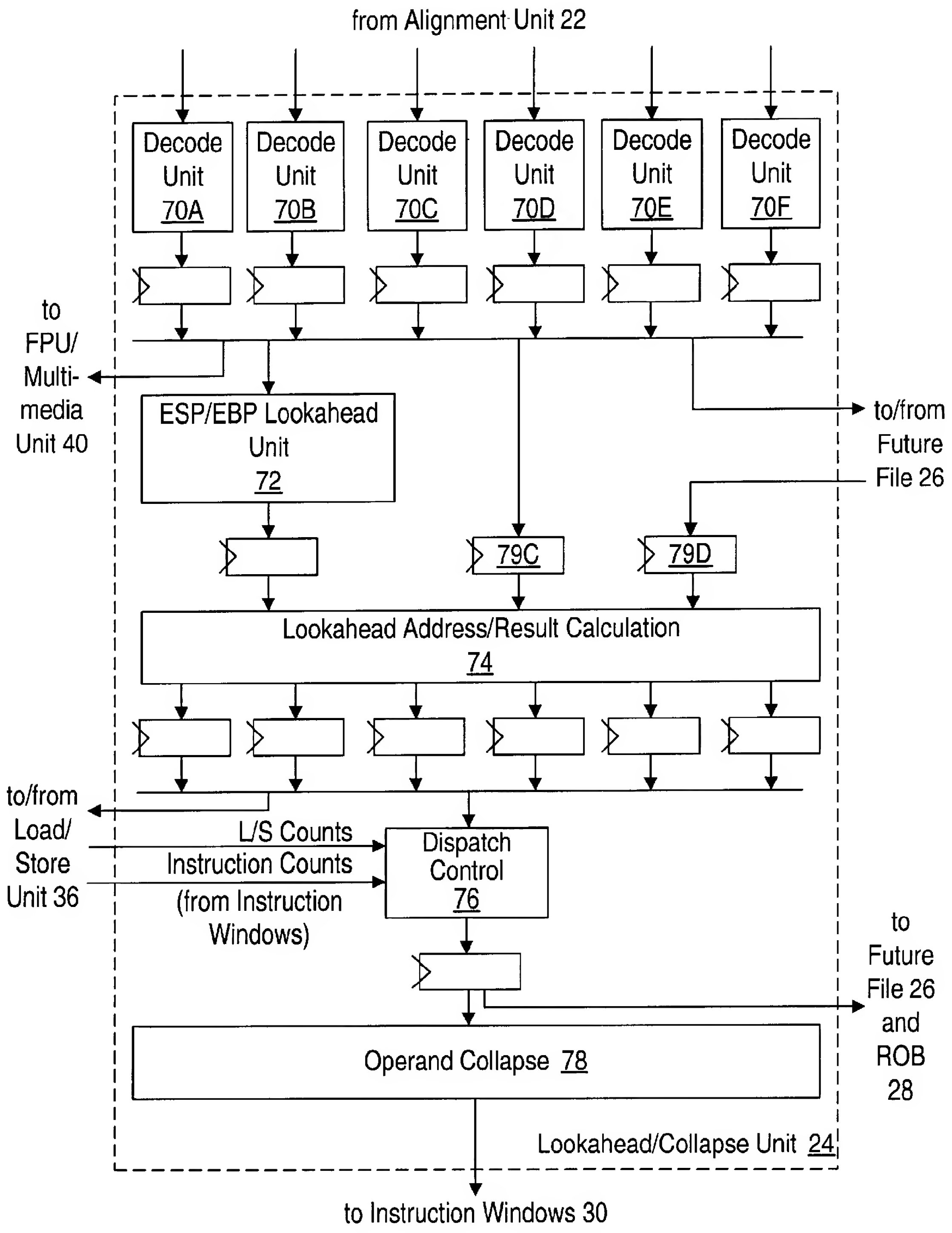

from Alignment Unit 22
Decode Unit 70A
Decode Unit 70B
Decode Unit 70C
Decode Unit 70D
Decode Unit 70E
Decode Unit 70F
to FPU/ Multi-media Unit 40
to/from Future File 26
ESP/EBP Lookahead Unit 72
79C
79D
Lookahead Address/Result Calculation 74
to/from Load/ Store Unit 36
L/S Counts
Instruction Counts (from Instruction Windows)
Dispatch Control 76
to Future File 26 and ROB 28
Operand Collapse 78
Lookahead/Collapse Unit 24
to Instruction Windows 30

FIG. 3

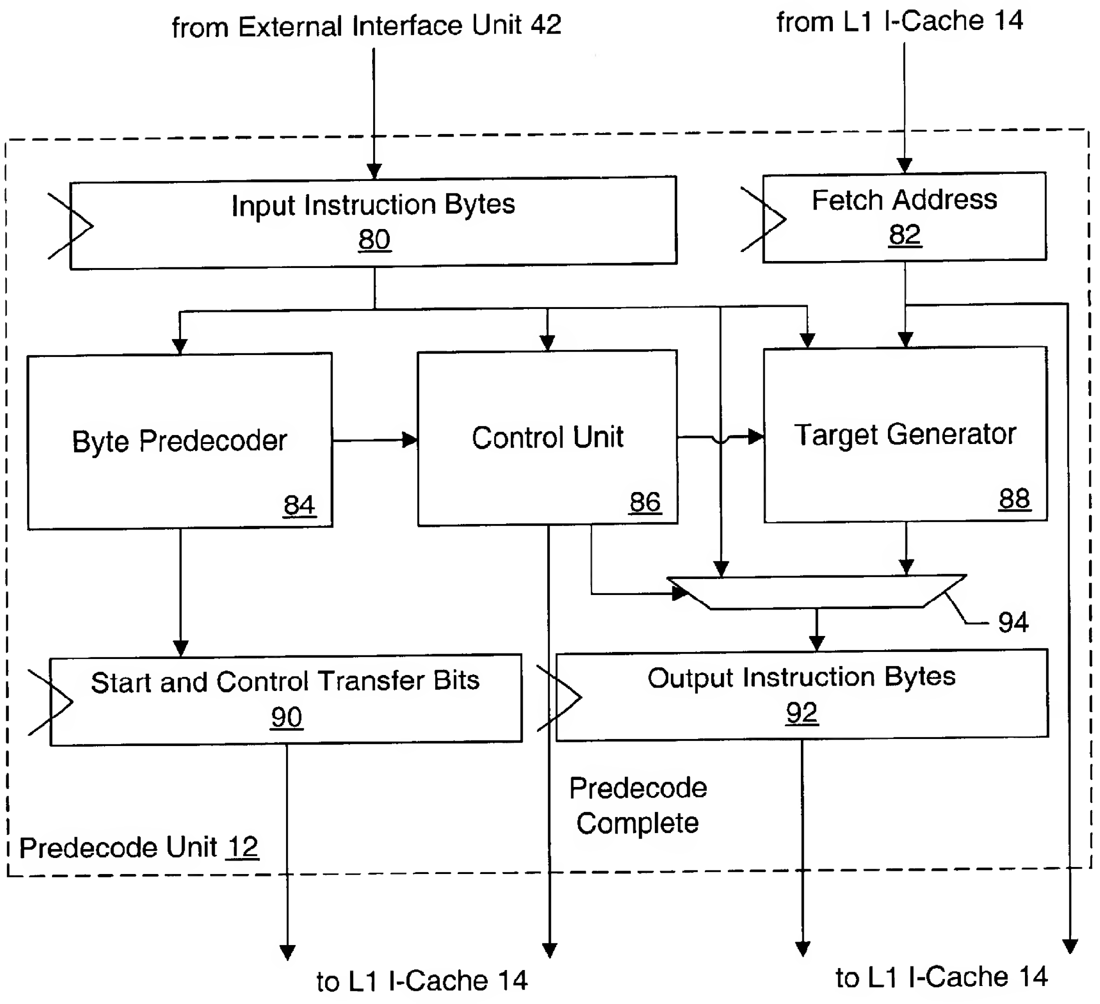

from External Interface Unit 42
from L1 I-Cache 14
Input Instruction Bytes
80
Fetch Address
82
Byte Predecoder
84
Control Unit
86
Target Generator
88
94
Start and Control Transfer Bits
90
Output Instruction Bytes
92
Predecode Complete
Predecode Unit 12
to L1 I-Cache 14
to L1 I-Cache 14

FIG. 4

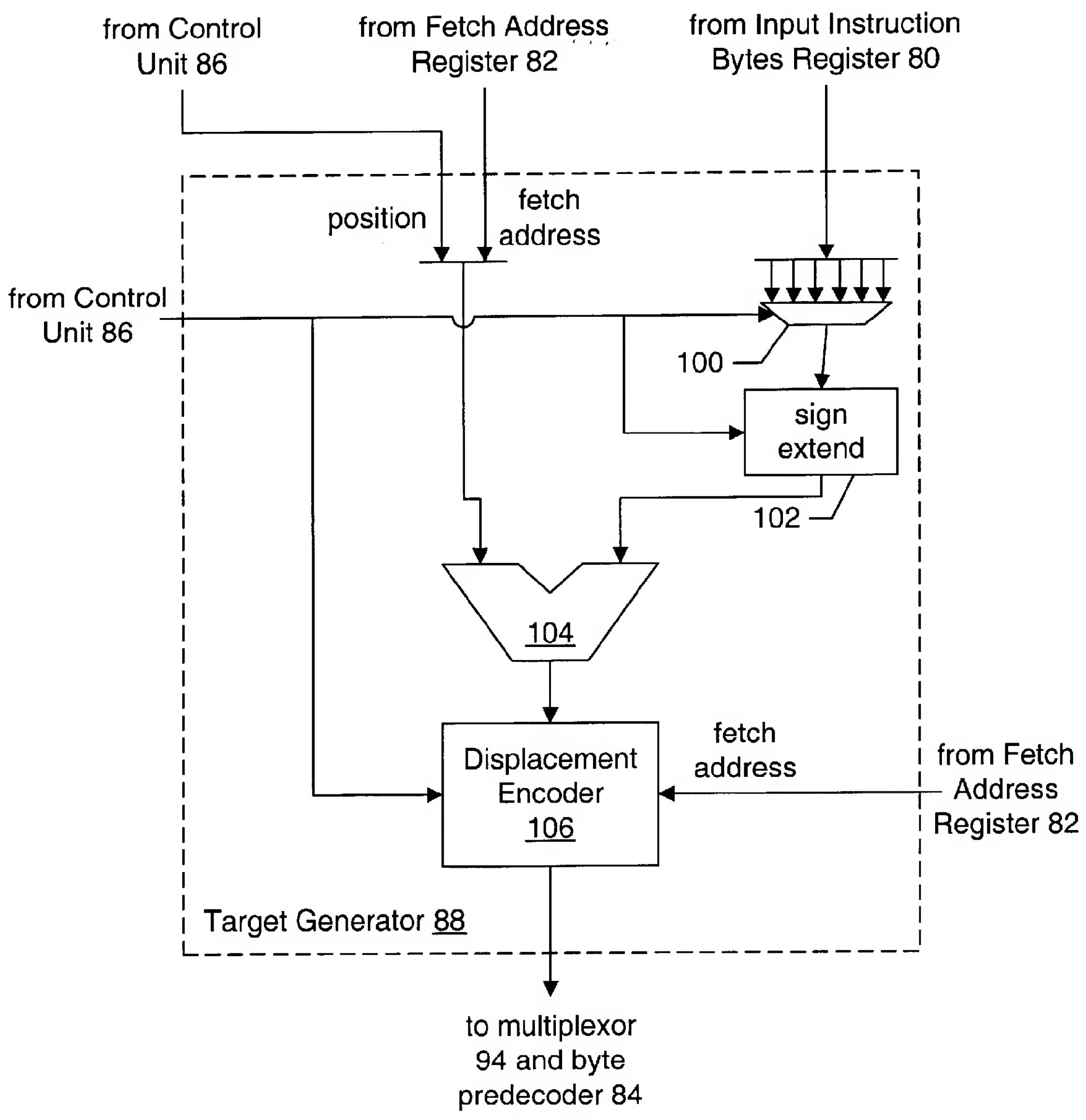

from Control Unit 86
from Fetch Address Register 82
from Input Instruction Bytes Register 80
position
fetch address
from Control Unit 86
100
sign extend
102
104
Displacement Encoder 106
fetch address
from Fetch Address Register 82
Target Generator 88
to multiplexor 94 and byte predecoder 84

FIG. 4A

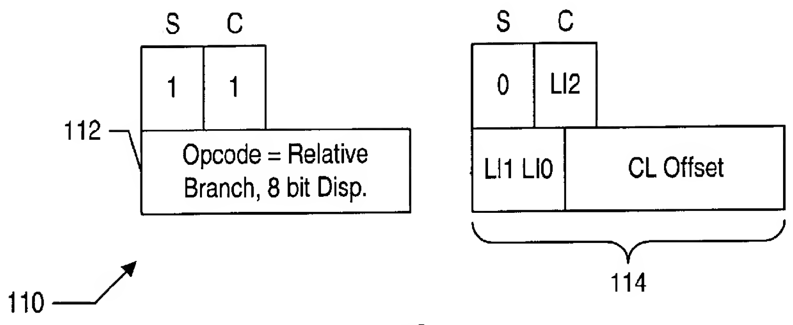


FIG. 5

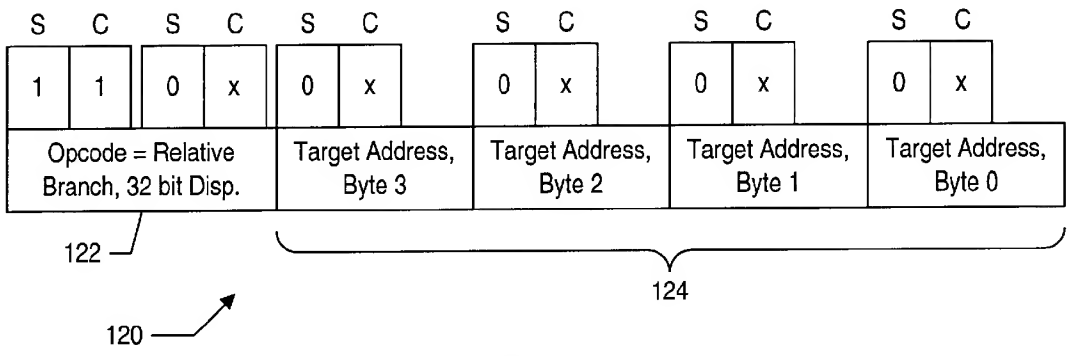


FIG. 6

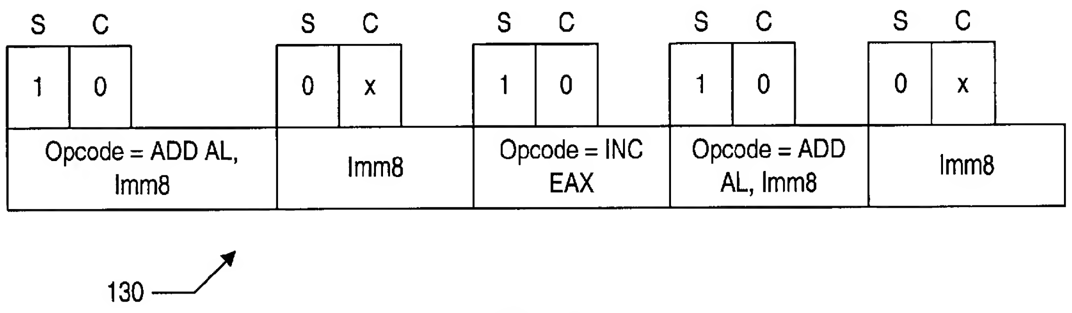


FIG. 7

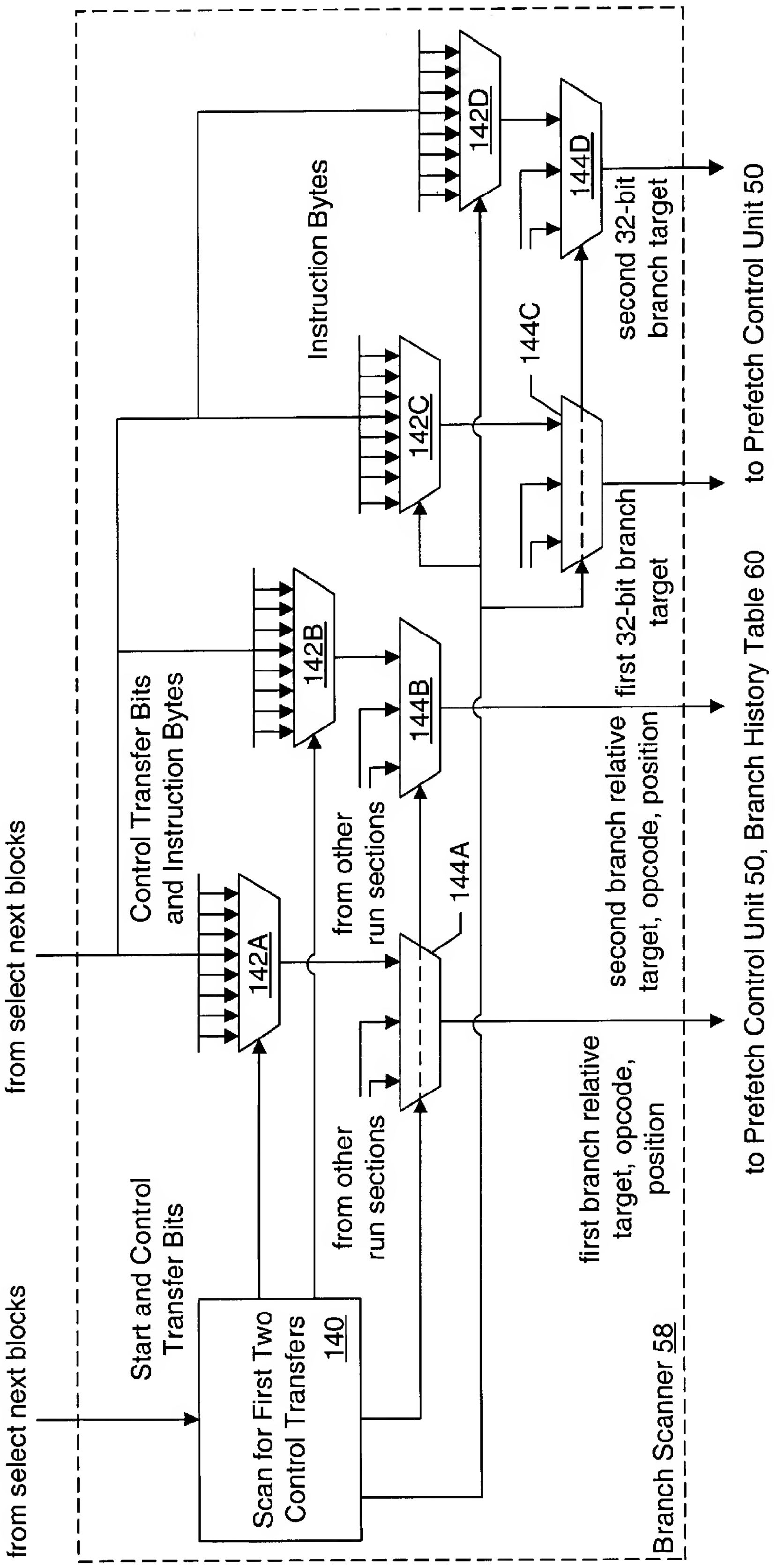

from select next blocks
from select next blocks
Start and Control Transfer Bits
Control Transfer Bits and Instruction Bytes
Instruction Bytes
Scan for First Two Control Transfers 140
142A
142B
142C
142D
144A
144B
144C
144D
from other run sections
from other run sections
first branch relative target, opcode, position
second branch relative target, opcode, position
first 32-bit branch target
second 32-bit branch target
Branch Scanner 58
to Prefetch Control Unit 50, Branch History Table 60
to Prefetch Control Unit 50

FIG. 8

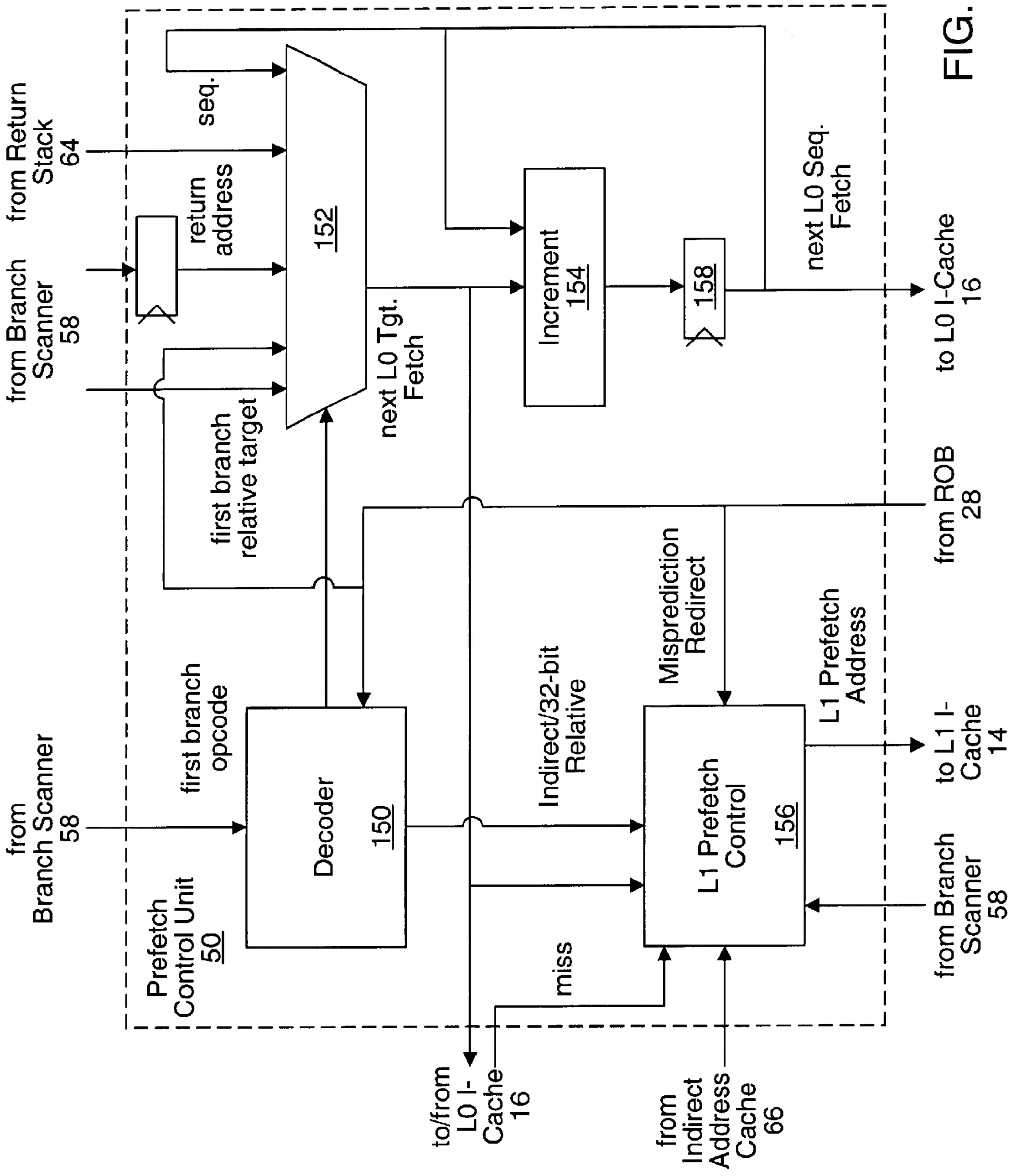

from Branch Scanner 58
from Branch Scanner 58
from Return Stack 64
Prefetch Control Unit 50
first branch opcode
Decoder 150
first branch relative target
return address
seq.
152
next L0 Tgt. Fetch
Indirect/32-bit Relative
miss
Increment 154
158
Misprediction Redirect
L1 Prefetch Control 156
next L0 Seq. Fetch
L1 Prefetch Address
to L0 I-Cache 16
from ROB 28
to L1 I-Cache 14
from Branch Scanner 58
to/from L0 I-Cache 16
from Indirect Address Cache 66

FIG. 9

Opcode	Selection	Note
Jcc rel 8 bit (7x, Ex)	first branch relative target	none
Return (Cx)	return address	none
Call/Jmp relative (Ex)	first branch relative target	Correct if not 8 bit displacement
Call/Jmp Indirect (Fx)	sequential	Signal L1 Prefetch Control
32 bit relative (0x)	first branch relative target	Signal L1 Prefetch Control, Correct fetch next cycle

160

FIG. 10

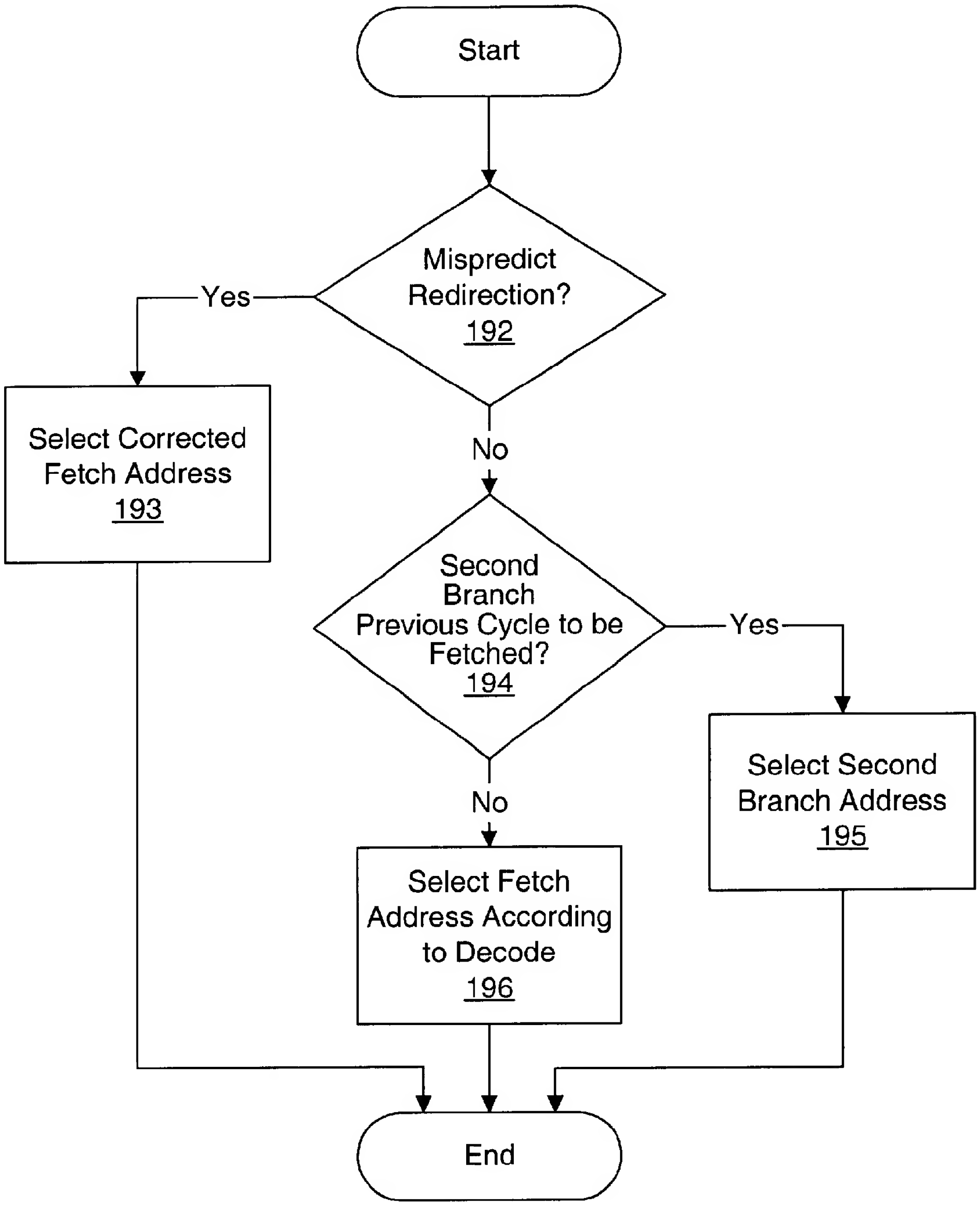

Start
Mispredict Redirection? 192
Yes
No
Select Corrected Fetch Address 193
Second Branch Previous Cycle to be Fetched? 194
Yes
No
Select Second Branch Address 195
Select Fetch Address According to Decode 196
End

FIG. 10A

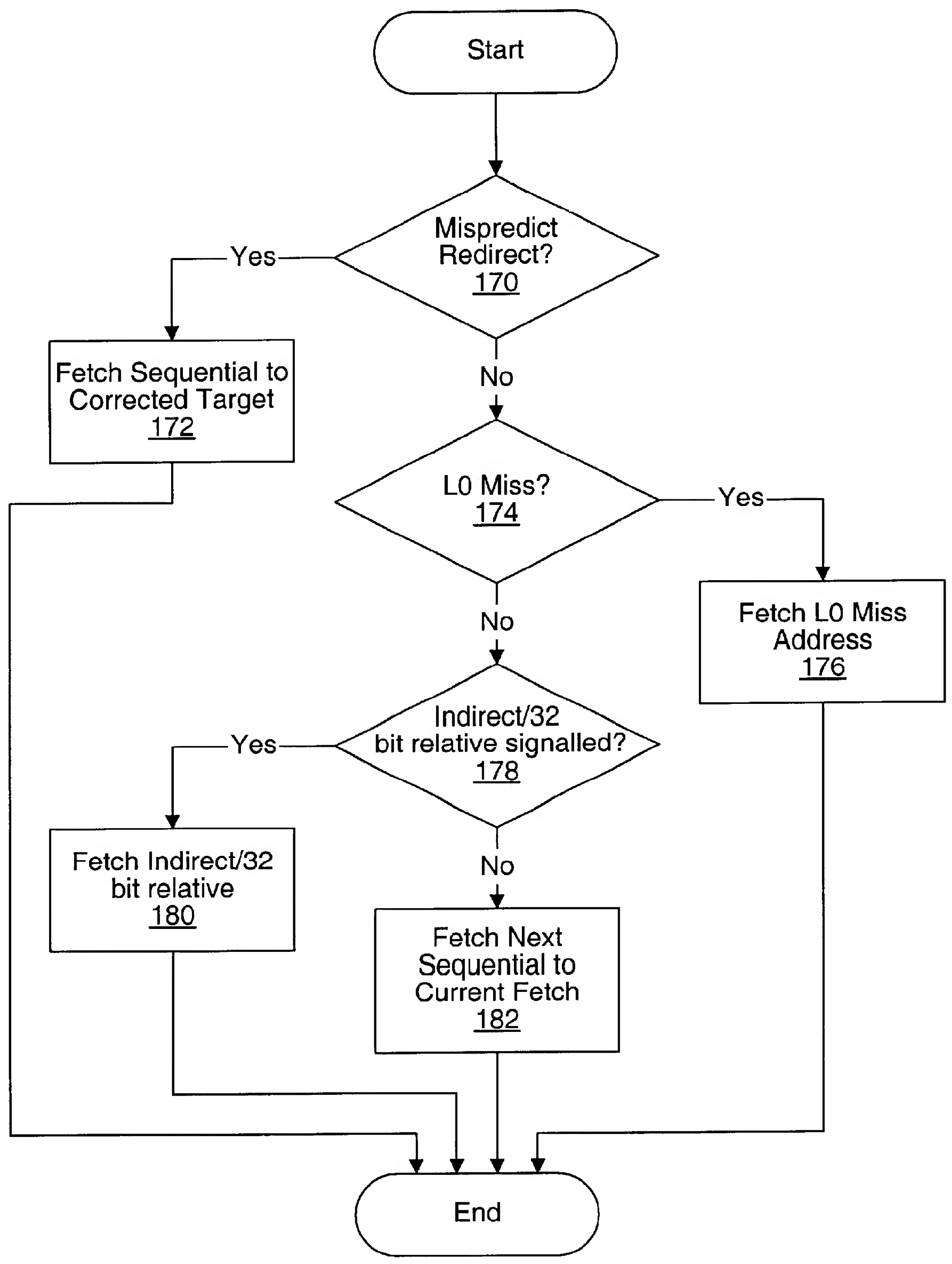

Start
Mispredict Redirect? 170
Yes
No
Fetch Sequential to Corrected Target 172
L0 Miss? 174
Yes
No
Fetch L0 Miss Address 176
Indirect/32 bit relative signalled? 178
Yes
No
Fetch Indirect/32 bit relative 180
Fetch Next Sequential to Current Fetch 182
End

FIG. 11

First Branch Target	First Branch Prediction	Second Branch Target	Second Branch Prediction	Result
Backward	Taken	x	x	Squash all instructions after first branch, Fetch target (L0)
Forward Small	Taken	Backward	Taken[1]	Squash instructions skipped by first and after 2nd, fetch 2nd Target (L0)
Forward Small	Taken	Forward Small	Taken[1]	Squash instructions skipped by each branch, fetch 2nd Target (L0)
Forward Small	Taken	Other	Taken[1]	Squash instructions skipped by each branch, fetch 2nd Target (L1)
Forward Small	Taken	x	Not Taken[1]	Squash instructions skipped by first branch, fetch sequential (L0)
Forward Large	Taken	x	x	Squash all instructions after first branch, Fetch target (L0)
32 bit relative or Indirect	Taken	x	x	Squash all instructions after first branch, Fetch target (L1)
Return	x	x	x	Squash all instructions after first branch, Fetch target (L0)
x	Not Taken	Backward	Taken[1]	Squash all instructions after second branch, Fetch target (L0)
x	Not Taken	Forward Small	Taken[1]	Squash instructions skipped by second branch, fetch 2nd Tgt (L0)
x	Not Taken	Other	Taken[1]	Squash instructions skipped by second branch, fetch 2nd Tgt (L1)
x	Not Taken	x	Not Taken[1]	fetch sequential (L0)

Note 1: Second Prediction may not be available in one clock cycle, so these results may be delayed by a clock cycle. May assume not taken in first clock cycle, and squash in second clock cycle with corrected fetch.

190

FIG. 12

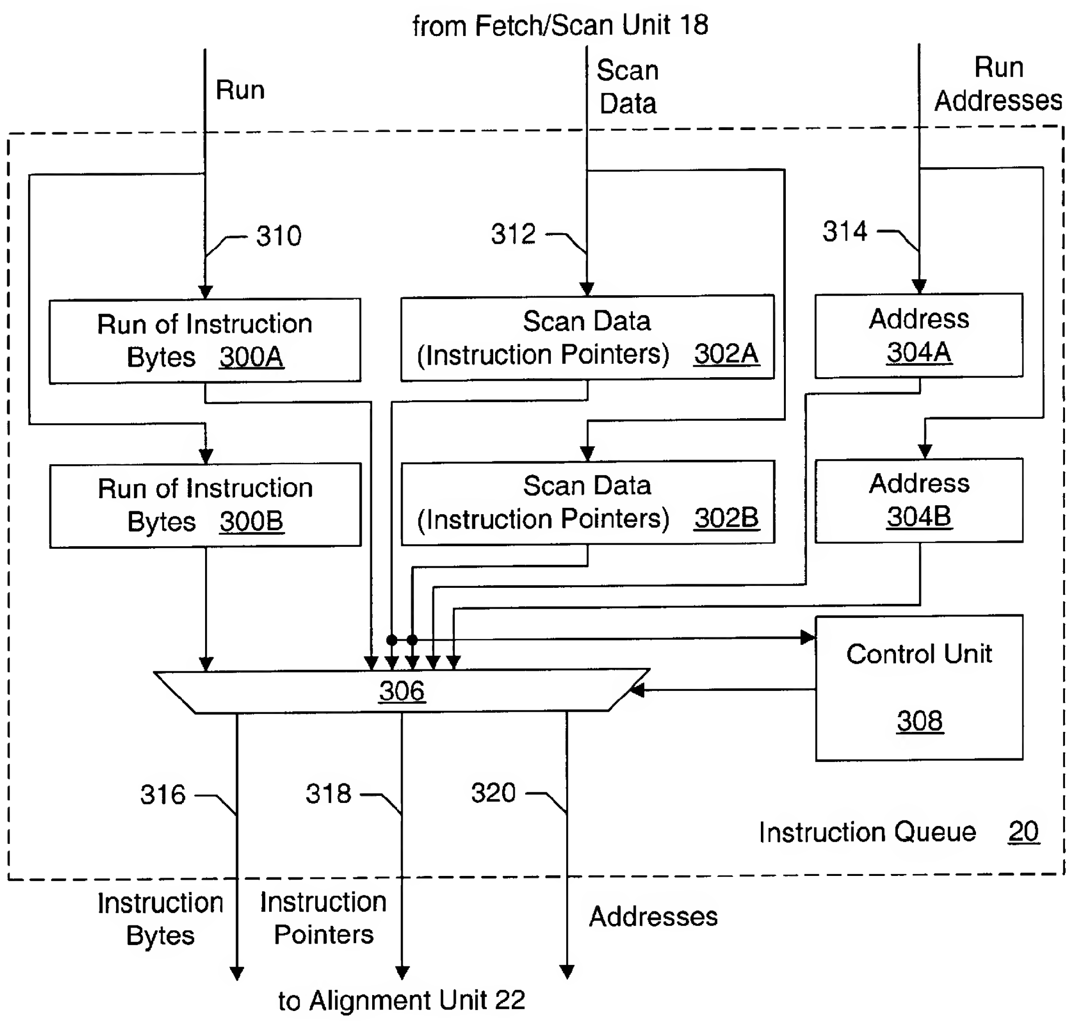

from Fetch/Scan Unit 18
Run
Scan Data
Run Addresses
310
312
314
Run of Instruction Bytes 300A
Scan Data (Instruction Pointers) 302A
Address 304A
Run of Instruction Bytes 300B
Scan Data (Instruction Pointers) 302B
Address 304B
Control Unit 308
306
316
318
320
Instruction Queue 20
Instruction Bytes
Instruction Pointers
Addresses
to Alignment Unit 22

FIG. 13

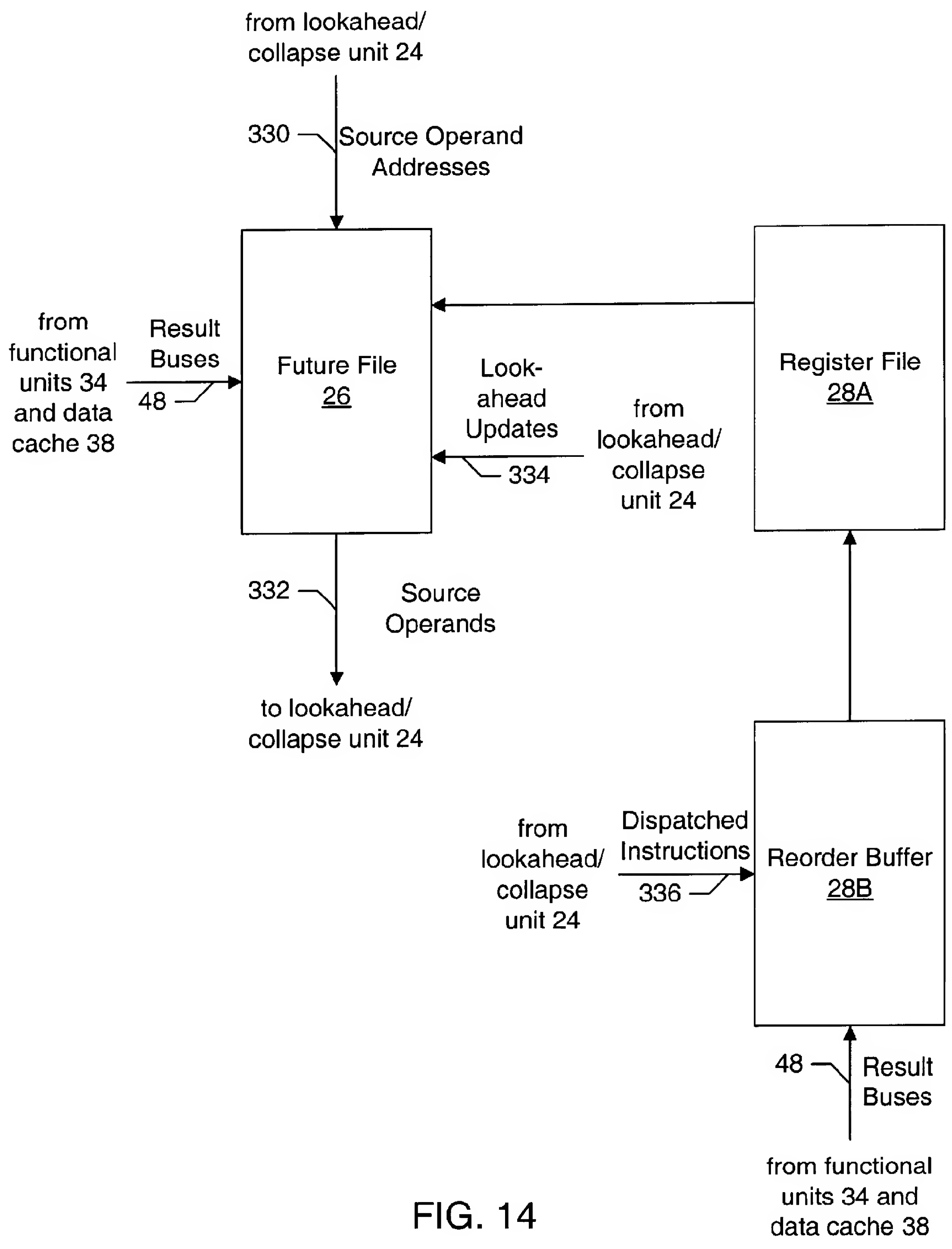

from lookahead/ collapse unit 24
330
Source Operand Addresses
from functional units 34 and data cache 38
Result Buses
48
Future File 26
Register File 28A
Look-ahead Updates
from lookahead/ collapse unit 24
334
332
Source Operands
to lookahead/ collapse unit 24
from lookahead/ collapse unit 24
Dispatched Instructions
336
Reorder Buffer 28B
48
Result Buses
from functional units 34 and data cache 38

FIG. 14

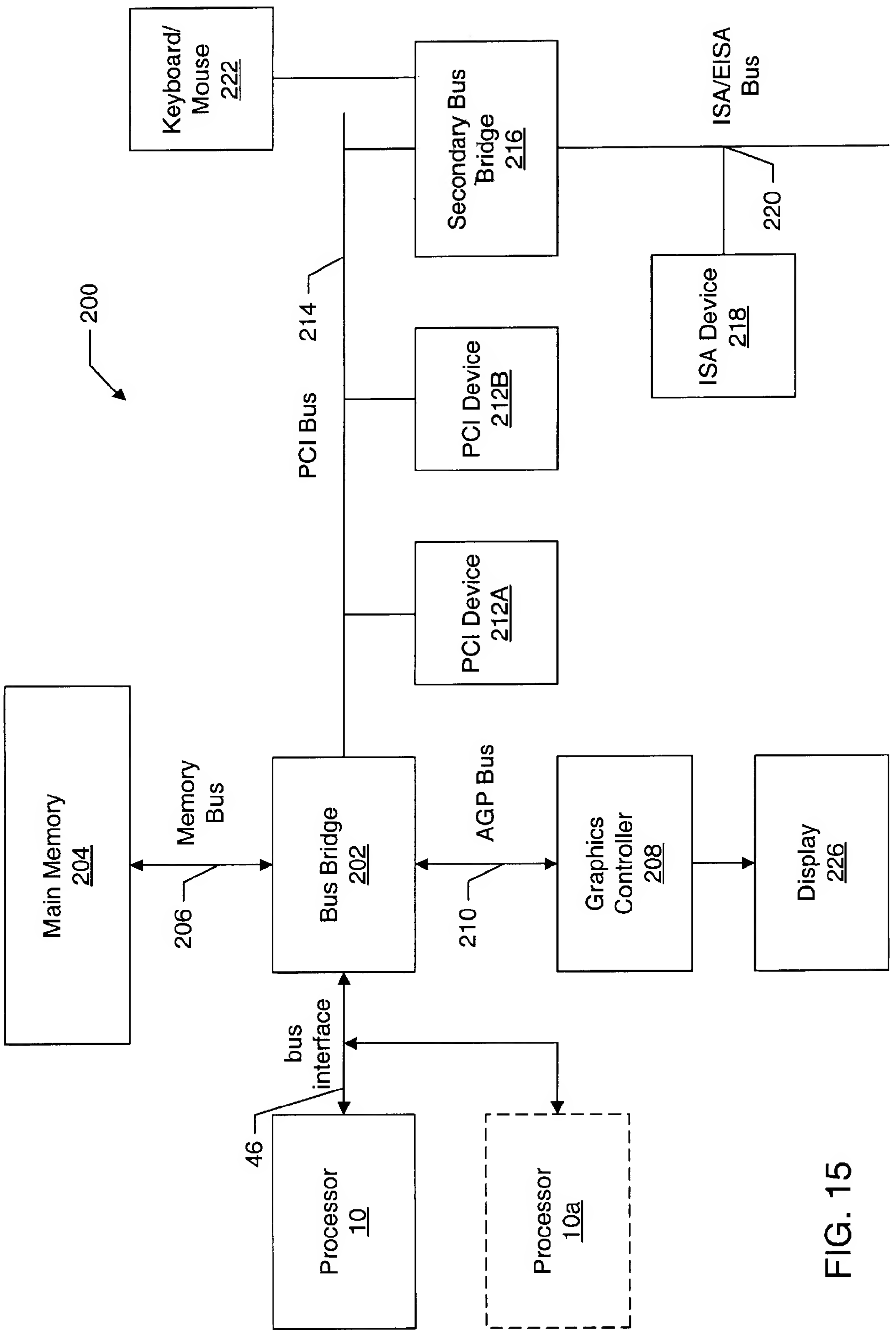

200
Keyboard/ Mouse 222
Secondary Bus Bridge 216
ISA/EISA Bus
220
ISA Device 218
214
PCI Bus
PCI Device 212B
PCI Device 212A
Main Memory 204
Memory Bus
206
Bus Bridge 202
AGP Bus
210
Graphics Controller 208
Display 226
bus interface
46
Processor 10
Processor 10a

FIG. 15

PROCESSOR CONFIGURED TO PREDECODE RELATIVE CONTROL TRANSFER INSTRUCTIONS AND REPLACE DISPLACEMENTS THEREIN WITH A TARGET ADDRESS

This Application is a continuation of U.S. patent application Ser. No. 09/065,681, now U.S. Pat. No. 6,167,506, filed Apr. 23, 1998, which claims benefit of priority to the Provisional Application Ser. No. 60/065,878, entitled "High Frequency, Wide Issue Microprocessor" filed on Nov. 17, 1997 by Witt. The Provisional Application is incorporated herein by reference in its entirety.

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention is related to the field of processors and, more particularly, to predecoding techniques within processors.

2. Description of the Related Art

Superscalar processors attempt to achieve high performance by dispatching and executing multiple instructions per clock cycle, and by operating at the shortest possible clock cycle time consistent with the design. To the extent that a given processor is successful at dispatching and/or executing multiple instructions per clock cycle, high performance may be realized. In order to increase the average number of instructions dispatched per clock cycle, processor designers have been designing superscalar processors which employ wider issue rates. A "wide issue" superscalar processor is capable of dispatching (or issuing) a larger maximum number of instructions per clock cycle than a "narrow issue" superscalar processor is capable of dispatching. During clock cycles in which a number of dispatchable instructions is greater than the narrow issue processor can handle, the wide issue processor may dispatch more instructions, thereby achieving a greater average number of instructions dispatched per clock cycle.

Many processors are designed to execute the x86 instruction set due to its widespread acceptance in the computer industry. For example, the K5 and K6 processors from Advanced Micro Devices, Inc., of Sunnyvale, Calif. implement the x86 instruction set. The x86 instruction set is a variable length instruction set in which various instructions occupy differing numbers of bytes in memory. The type of instruction, as well as the addressing modes selected for a particular instruction encoding, may affect the number of bytes occupied by that particular instruction encoding. Variable length instruction sets, such as the x86 instruction set, minimize the amount of memory needed to store a particular program by only occupying the number of bytes needed for each instruction. In contrast, many RISC architectures employ fixed length instruction sets in which each instruction occupies a fixed, predetermined number of bytes.

Unfortunately, variable length instruction sets complicate the design of wide issue processors. For a wide issue processor to be effective, the processor must be able to identify large numbers of instructions concurrently and rapidly within a code sequence in order to provide sufficient instructions to the instruction dispatch hardware. Because the location of each variable length instruction within a code sequence is dependent upon the preceding instructions, rapid identification of instructions is difficult. If a sufficient number of instructions cannot be identified, the wide issue structure may not result in significant performance gains. Therefore, a processor which provides rapid and concurrent identification of instructions for dispatch is needed.

Another feature which is important to the performance achievable by wide issue superscalar processors is the accuracy and effectiveness of its branch prediction mechanism. As used herein, the branch prediction mechanism refers to the hardware which detects control transfer instructions within the instructions being identified for dispatch and which predicts the next fetch address resulting from the execution of the identified control transfer instructions. Generally, a "control transfer" instruction is an instruction which, when executed, specifies the address from which the next instruction to be executed is fetched. Jump instructions are an example of control transfer instructions. A jump instruction specifies a target address different than the address of the byte immediately following the jump instruction (the "sequential address"). Unconditional jump instructions always cause the next instruction to be fetched to be the instruction at the target address, while conditional jump instructions cause the next instruction be fetched to be either the instruction at the target address or the instruction at the sequential address responsive to an execution result of a previous instruction (for example, by specifying a condition flag set via instruction execution). Other types of instructions besides jump instructions may also be control transfer instructions. For example, subroutine call and return instructions may cause stack manipulations in addition to specifying the next fetch address. Many of these additional types of control transfer instructions include a jump operation (either conditional or unconditional) as well as additional instruction operations.

Control transfer instructions may specify the target address in a variety of ways. "Relative" control transfer instructions include a value (either directly or indirectly) which is to be added to an address corresponding to the relative control transfer instruction in order to generate the target address. The address to which the value is added depends upon the instruction set definition. For x86 control transfer instructions, the address of the byte immediately following the control transfer instruction is the address to which the value is added. Other instruction sets may specifying adding the value to the address of the control transfer instruction itself. For relative control transfer instructions which directly specify the value to be added, an instruction field is included for storing the value and the value is referred to as a "displacement".

On the other hand, "absolute" control transfer instructions specify the target address itself (again, either directly or indirectly). Absolute control transfer instructions therefore do not require an address corresponding to the control transfer instruction to determine the target address. Control transfer instructions which specify the target address indirectly (e.g. via one or more register or memory operands) are referred to as "indirect" control transfer instructions.

Because of the variety of available control transfer instructions, the branch prediction mechanism may be quite complex. However, because control transfer instructions occur frequently in many program sequences, wide issue processors have a need for a highly effective (e.g. both accurate and rapid) branch prediction mechanism. If the branch prediction mechanism is not highly accurate, the wide issue processor may issue a large number of instructions per clock cycle but may ultimately cancel many of the issued instructions due to branch mispredictions. On the other hand, the number of clock cycles used by the branch prediction mechanism to generate a target address needs to be minimized to allow for the instructions that the target address to be fetched.

The term "branch instruction" is used herein to be synonymous with "control transfer instruction".

SUMMARY OF THE INVENTION

The problems outlined above are in large part solved by a processor in accordance with the present invention. The processor is configured to predecode instruction bytes prior to their storage within an instruction cache. During the predecoding, relative branch instructions are detected. The displacement included within the relative branch instruction is added to the address corresponding to the relative branch instruction, thereby generating the target address. The processor replaces the displacement field of the relative branch instruction with an encoding of the target address, and stores the modified relative branch instruction in the instruction cache. Advantageously, the branch prediction mechanism employed by the processor may more rapidly generate the target address corresponding to relative branch instructions. The branch prediction mechanism may simply select the target address from the displacement field of the relative branch instruction instead of performing an addition to generate the target address. The rapidly generated target address may be provided to the instruction cache for fetching instructions more quickly than might otherwise be achieved. The amount of time elapsing between fetching a branch instruction and generating the corresponding target address may advantageously be reduced. Accordingly, the branch prediction mechanism may operate more efficiently, and hence processor performance may be increased through more rapid fetching of instructions stored at the target address. Superscalar processors may thereby support wider issue rates by fetching a larger number of instructions in a given period of time.

In one embodiment, relative branch instructions having eight bit and 32-bit displacement fields are included in the instruction set executed by the processor. Additionally, the processor employs predecode information (stored in the instruction cache with the corresponding instruction bytes) including a start bit and a control transfer bit corresponding to each instruction byte. The combination of the start bit indicating that the byte is the start of an instruction and the corresponding control transfer bit identifies the instruction as either a branch instruction or a non-branch instruction. For relative branch instructions including an eight bit displacement, the control transfer bit corresponding to the displacement field is used in conjunction with the displacement field to store the encoded target address. The encoded target address includes a cache line offset portion and a relative cache line portion identifying the target cache line as a function of the cache line storing the relative branch instruction. Thirty-two bit displacement fields store the entirety of the target address, and hence the encoded target address comprises the target address. Other embodiments than the one described above are contemplated.

Broadly speaking, the present invention contemplates a processor comprising a predecode unit and an instruction cache. The predecode unit is configured to predecode a plurality of instruction bytes received by the processor. Upon predecoding a relative control transfer instruction comprising a displacement, the predecode unit adds an address to the displacement to generate a target address corresponding to the relative control transfer instruction. Additionally, the predecode unit is configured to replace the displacement within the relative control transfer instruction with at least a portion of the target address. Coupled to the predecode unit, the instruction cache is configured to store the plurality of instruction bytes including the relative control transfer instruction with the displacement replaced by the portion of the target address.

The present invention further contemplates a method for generating a target address for a relative control transfer instruction. A plurality of instruction bytes including the relative transfer instruction are predecoded to detect the presence of the relative control transfer instruction. An address is added to a displacement included in the relative control transfer instruction, thereby generating the target address. The displacement is replaced within the relative control transfer instruction with an encoding indicative of the target address. The plurality of instruction bytes including the relative control transfer instruction is stored in an instruction cache, with the displacement replaced by the encoding.

Moreover, the present invention contemplates a predecode unit comprising a decoder and a target generator. The decoder is configured to decode a plurality of instruction bytes and to identify a relative control transfer instruction therein. The target generator is configured to add a displacement selected from the relative control transfer instruction to an address, thereby generating a target address corresponding to the relative control transfer instruction, and is further configured to generate an encoding of the target address with which the predecode unit replaces the displacement within the relative control transfer instruction.

The present invention still further contemplates a computer system comprising a processor, a memory, and an input/output (I/O) device. The processor is configured to predecode a plurality of instruction bytes received by the processor. Upon predecoding a relative control transfer instruction comprising a displacement, the processor is configured to add an address to the displacement to generate a target address corresponding to the relative control transfer instruction. Additionally, the processor is configured to replace the displacement within the relative control transfer instruction with at least a portion of the target address. Coupled to the processor, the memory is configured to store the plurality of instruction bytes and to provide the instruction bytes to the processor. The I/O device is configured to transfer data between the computer system and another computer system coupled to the I/O device.

BRIEF DESCRIPTION OF THE DRAWINGS

Other objects and advantages of the invention will become apparent upon reading the following detailed description and upon reference to the accompanying drawings in which:

FIG. **1** is a block diagram of one embodiment of a superscalar processor.

FIG. **2** is a block diagram of one embodiment of a fetch/scan unit shown in FIG. **1**.

FIG. **3** is a block diagram of one embodiment of a decode and lookahead/collapse unit shown in FIG. **1**.

FIG. **4** is a block diagram of one embodiment of a predecode unit shown in FIG. **1**.

FIG. **4A** is a block diagram of one embodiment of a target generator shown in FIG. **4**.

FIG. **5** is a diagram illustrating a control transfer instruction having an 8-bit offset and the corresponding predecode information according to one embodiment of the processor shown in FIG. 1.

FIG. **6** is a diagram illustrating a control transfer instruction having a 32-bit offset and the corresponding predecode information according to one embodiment of the processor shown in FIG. 1.

FIG. **7** is a diagram illustrating several non-control transfer instructions and the corresponding predecode information according to one embodiment of the processor shown in FIG. **1**.

FIG. **8** is a block diagram of one embodiment of a branch scanner shown in FIG. **2**.

FIG. **9** is a block diagram of one embodiment of a prefetch control unit shown in FIG. **2**.

FIG. **10** is a truth table for one embodiment of the decoder shown in FIG. **9**.

FIG. **10A** is a flowchart illustrating operation of one embodiment of the decoder shown in FIG. **9**.

FIG. **11** is a flowchart illustrating operation of one embodiment of the L1 prefetch control unit shown in FIG. **9**.

FIG. **12** is a table illustrating instruction fetch and dispatch results for one embodiment of the processor shown in FIG. **1** in which up to two branch instructions are predicted per clock cycle.

FIG. **13** is a block diagram of one embodiment of an instruction queue illustrated in FIG. **1**.

FIG. **14** is a block diagram of one embodiment of a future file, register file, and reorder buffer shown in FIG. **1**.

FIG. **15** is a block diagram of one embodiment of a computer system including the processor shown in FIG. **1**.

While the invention is susceptible to various modifications and alternative forms, specific embodiments thereof are shown by way of example in the drawings and will herein be described in detail. It should be understood, however, that the drawings and detailed description thereto are not intended to limit the invention to the particular form disclosed, but on the contrary, the intention is to cover all modifications, equivalents and alternatives falling within the spirit and scope of the present invention as defined by the appended claims.

DETAILED DESCRIPTION OF THE INVENTION

Turning now to FIG. **1**, a block diagram of one embodiment of a superscalar processor **10** is shown. Other embodiments are possible and contemplated. In the embodiment shown in FIG. **1**, processor **10** includes a predecode unit **12**, an L1 I-cache **14**, an L0 I-cache **16**, a fetch/scan unit **18**, an instruction queue **20**, an alignment unit **22**, a lookahead/collapse unit **24**, a future file **26**, a reorder buffer/register file **28**, a first instruction window **30A**, a second instruction window **30B**, a plurality of functional units **32A**, **32B**, **32C**, and **32D**, a plurality of address generation units **34A**, **34B**, **34C**, and **34D**, a load/store unit **36**, an L1 D-cache **38**, an FPU/multimedia unit **40**, and an external interface unit **42**. Elements referred to herein by a particular reference number followed by various letters will be collectively referred to using the reference number alone. For example, functional units **32A**, **32B**, **32C**, and **32D** will be collectively referred to as functional units **32**.

In the embodiment of FIG. **1**, external interface unit **42** is coupled to predecode unit **12**, L1 D-cache **38**, an L2 interface **44**, and a bus interface **46**. Predecode unit **12** is further coupled to L1 I-cache **14**. L1 I-cache **14** is coupled to L0 I-cache **16** and to fetch/scan unit **18**. Fetch/scan unit **18** is also coupled to L0 I-cache **16** and to instruction queue **20**. Instruction queue **20** is coupled to alignment unit **22**, which is further coupled to lookahead/collapse unit **24**. Lookahead/collapse unit **24** is further coupled to future file **26**, reorder buffer/register file **28**, load/store unit **36**, first instruction window **30A**, second instruction window **30B**, and FPU/multimedia unit **40**. FPU/multimedia unit **40** is coupled to load/store unit **36** and to reorder buffer/register file **28**. Load/store unit **36** is coupled to L1 D-cache **38**. First

instruction window **30A** is coupled to functional units **32A–32B** and to address generation units **34A–34B**. Similarly, second instruction window **30B** is coupled to functional units **32C–32D** and address generation units **34C–34D**. Each of L1 D-cache **38**, functional units **32**, and address generation units **34** are coupled to a plurality of result buses **48** which are further coupled to load/store unit **36**, first instruction window **30A**, second instruction window **30B**, reorder buffer/register file **28**, and future file **26**.

Predecode unit **12** receives instruction bytes fetched by external interface unit **42** and predecodes the instruction bytes prior to their storage within L1 I-cache **14**. Predecode information generated by predecode unit **12** is stored in L1 I-cache **14** as well. Generally, predecode information is provided to aid in the identification of instruction features which may be useful during the fetch and issue of instructions but which may be difficult to generate rapidly during the fetch and issue operation. The term "predecode", as used herein, refers to decoding instructions to generate predecode information which is later stored along with the instruction bytes being decoded in an instruction cache (e.g. L1 I-cache **14** and/or L0 I-cache **16**).

In one embodiment, processor **10** employs two bits of predecode information per instruction byte. One of the bits, referred to as the "start bit", indicates whether or not the instruction byte is the initial byte of an instruction. When a group of instruction bytes is fetched, the corresponding set of start bits identifies the boundaries between instructions within the group of instruction bytes. Accordingly, multiple instructions may be concurrently selected from the group of instruction bytes by scanning the corresponding start bits. While start bits are used to locate instruction boundaries by identifying the initial byte of each instruction, end bits could alternatively be used to locate instruction boundaries by identifying the final byte of each instruction.

The second predecode bit used in this embodiment, referred to as the "control transfer" bit, identifies which instructions are branch instructions. The control transfer bit corresponding to the initial byte of an instruction indicates whether or not the instruction is a branch instruction. The control transfer bit corresponding to subsequent bytes of the instruction is a don't care except for relative branch instructions having a small displacement field. According to one particular embodiment, the small displacement field is an 8 bit field. Generally, a "small displacement field" refers to a displacement field having fewer bits than the target address generated by branch instructions. For relative branch instructions having small displacement fields, the control transfer bit corresponding to the displacement byte is used as described below.

In addition to generating predecode information corresponding to the instruction bytes, predecode unit **12** is configured to recode the displacement field of relative branch instructions to actually store the target address in the present embodiment. In other words, predecode unit **12** adds the displacement of the relative branch instruction to the address corresponding to the relative branch instruction as defined by the instruction set employed by processor **10**. The resulting target address is encoded into the displacement field as a replacement for the displacement, and the updated displacement field is stored into L1 I-cache **14** instead of the original displacement field. Target address generation is simplified by precomputing relative target addresses, and hence the branch prediction mechanism may operate more efficiently.

In one embodiment of processor **10** which employs the x86 instruction set, predecode unit **12** is configured to recode eight bit and 32 bit displacement fields. The 32 bit displacement fields may store the entirety of the target address. On the other hand, the eight bit displacement field is encoded. More particularly, the eight bit displacement field and corresponding control transfer predecode bit is divided into a cache line offset portion and a relative cache line portion. The cache line offset portion is the cache line offset portion of the target address. The relative cache line portion defines the cache line identified by the target address (the "target cache line") in terms of a number of cache lines above or below the cache line storing the relative branch instruction. A first cache line is above a second cache line if each byte within the first cache line is stored at an address which is numerically greater than the addresses at which the bytes within the second cache line are stored. Conversely, a first cache line is below the second cache line if each byte within the first cache line is stored at an address which is numerically less than the addresses which the bytes within a second cache line are stored. A signed eight bit displacement specifies an address which is +/−128 bytes of the address corresponding to the branch instruction. Accordingly, the number of above and below cache lines which can be reached by a relative branch instruction having an eight bit displacement is limited. The relative cache line portion encodes this limited set of above and below cache lines.

Tables 1 and 2 below illustrates an exemplary encoding of the predecode information corresponding to a byte in accordance with one embodiment of processor **10**.

TABLE 1

Predecode Encoding

Start Bit	Control Transfer Bit	Meaning
1	0	Start byte of an instruction which is not a branch.
1	1	Start byte of a branch instruction.
0	x	Not an instruction boundary. Control Transfer Bit corresponding to displacement is used on 8-bit relative branches to encode target address as shown in Table 2 below.

TABLE 2

Target Address Encoding

Control Transfer Bit	Displacement Byte Most Significant Bits (binary)	Meaning
0	00	Within Current Cache Line
0	01	One Cache Line Above
0	10	Two Cache Lines Above
1	01	One Cache Line Below
1	10	Two Cache Lines Below

Note: Remaining displacement byte bits are the offset within the target cache line. Control Transfer Bit is effectively a direction, and the most significant bits of the displacement byte are the number of cache lines.

Predecode unit **12** conveys the received instruction bytes and corresponding predecode information to L1 I-cache **14** for storage. L1 I-cache **14** is a high speed cache memory for storing instruction bytes and predecode information. L1 I-cache **14** may employ any suitable configuration, including direct mapped and set associative configurations. In one particular embodiment, L1 I-cache **14** is a 128 KB, two way set associative cache employing 64 byte cache lines. L1 I-cache **14** includes additional storage for the predecode information corresponding to the instruction bytes stored therein. The additional storage is organized similar to the

instruction bytes storage. As used herein, the term "cache line" refers to the unit of allocation of storage in a particular cache. Generally, the bytes within a cache line are manipulated (i.e. allocated and deallocated) by the cache as a unit.

In one embodiment, L1 I-cache **14** is linearly addressed and physically tagged. A cache is linearly addressed if at least one of the address bits used to index the cache is a linear address bit which is subsequently translated to a physical address bit. The tags of a linearly address/physically tagged cache include each translated bit in addition to the bits not used to index. As specified by the x86 architecture, instructions are defined to generate logical addresses which are translated through a segmentation translation mechanism to a linear address and further translated through a page translation mechanism to a physical address. It is becoming increasingly common to employ flat addressing mode, in which the logical address and corresponding linear address are equal. Processor **10** may be configured to assume flat addressing mode. Accordingly, fetch addresses, target addresses, etc. as generated by executing instructions are linear addresses. In order to determine if a hit is detected in L1 I-cache **14**, the linear address presented thereto by fetch/scan unit **18** is translated using a translation lookaside buffer (TLB) to a corresponding physical address which is compared to the physical tags from the indexed cache lines to determine a hit/miss. When flat addressing mode is not used, processor **10** may still execute code but additional clock cycles may be used to generate linear addresses from logical addresses.

L0 I-cache **16** is also a high speed cache memory for storing instruction bytes. Because L1 I-cache **14** is large, the access time of L1 I-cache **14** may be large. In one particular embodiment, L1 I-cache **14** uses a two clock cycle access time. In order to allow for single cycle fetch access, L0 I-cache **16** is employed. L0 I-cache **16** is comparably smaller than L1 I-cache **14**, and hence may support a more rapid access time. In one particular embodiment, L0 I-cache **16** is a 512 byte fully associative cache. Similar to L1 I-cache **14**, L0 I-cache **16** is configured to store cache lines of instruction bytes and corresponding predecode information (e.g. 512 bytes stores eight 64 byte cache lines and corresponding predecode data is stored in additional storage). In one embodiment, L0 I-cache **16** may be linearly addressed and linearly tagged.

Fetch/scan unit **18** is configured to generate fetch addresses for L0 I-cache **16** and prefetch addresses for L1 I-cache **14**. Instructions fetched from L0 I-cache **16** are scanned by fetch/scan unit **18** to identify instructions for dispatch as well as to locate branch instructions and to form branch predictions corresponding to the located branch instructions. Instruction scan information and corresponding instruction bytes are stored into instruction queue **20** by fetch/scan unit **18**. Additionally, the identified branch instructions and branch predictions are used to generate subsequent fetch addresses for L0 I-cache **16**.

Fetch/scan unit **18** employs a prefetch algorithm to attempt to prefetch cache lines from L1 I-cache **14** to L0 I-cache **16** prior to the prefetched cache lines being fetched by fetch scan unit **18** for dispatch into processor **10**. Any suitable prefetch algorithm may be used. In one embodiment, fetch/scan unit **18** is configured to prefetch the next sequential cache line to a cache line fetched from L0 I-cache **16** during a particular clock cycle unless: (i) a branch misprediction is signalled; (ii) an L0 I-cache miss is detected; or (iii) a target address is generated which is assumed to miss L0 I-cache **16**. In one particular embodiment, relative branch instructions employing 32-bit displacements and branch instructions employing indirect target address generation are assumed to miss L0 I-cache **16**. For case (i), fetch/scan unit **18** prefetches the cache line sequential to the corrected fetch address. For cases (ii) and (iii), fetch/scan unit **18** prefetches the corresponding miss or target address.

Fetch/scan unit **18** employs an aggressive branch prediction mechanism in attempt to fetch larger "runs" of instructions during a clock cycle. As used herein, a "run" of instructions is a set of one or more instructions predicted to be executed in the sequence specified within the set. For example, fetch/scan unit **18** may fetch runs of 24 instruction bytes from L0 I-cache **16**. Each run is divided into several sections which fetch/scan unit **18** scans in parallel to identify branch instructions and to generate instruction scan information for instruction queue **20**. According to one embodiment, fetch/scan unit **18** attempts to predict up to two branch instructions per clock cycle in order support large instruction runs.

Instruction queue **20** is configured to store instruction bytes provided by fetch/scan unit **18** for subsequent dispatch. Instruction queue **20** may operate as a first-in, first-out (FIFO) buffer. In one embodiment, instruction queue **20** is configured to store multiple entries, each entry comprising: a run of instructions, scan data identifying up to five instructions within each section of the run, and addresses corresponding to each section of the run. Additionally, instruction queue **20** may be configured to select up to six instructions within up to four consecutive run sections for presentation to alignment unit **22**. Instruction queue **20** may, for example, employ 2–3 entries. An exemplary embodiment of instruction queue **20** is illustrated below in FIG. **13**.

Alignment unit **22** is configured to route instructions identified by instruction queue **20** to a set of issue positions within lookahead/collapse unit **24**. In other words, alignment unit **22** selects the bytes which form each instruction from the run sections provided by instruction queue **20** responsive to the scan information provided by instruction queue **20**. The instructions are provided into the issue positions in program order (i.e. the instruction which is first in program order is provided to the first issue position, the second instruction in program order is provided to the second issue position, etc.).

Lookahead/collapse unit **24** decodes the instructions provided by alignment unit **22**. FPU/multimedia instructions detected by lookahead/collapse unit **24** are routed to FPU/multimedia unit **40**. Other instructions are routed to first instruction window **30A**, second instruction window **30B**, and/or load/store unit **36**. In one embodiment, a particular instruction is routed to one of first instruction window **30A** or second instruction window **30B** based upon the issue position to which the instruction was aligned by alignment unit **22**. According to one particular embodiment, instructions from alternate issue positions are routed to alternate instruction windows **30A** and **30B**. For example, instructions from issue positions zero, two, and four may be routed to the first instruction window **30A** and instructions from issue positions one, three, and five may be routed to the second instruction window **30B**. Instructions which include a memory operation are also routed to load/store unit **36** for access to L1 D-cache **38**.

Additionally, lookahead/collapse unit **24** attempts to generate lookahead addresses or execution results for certain types of instructions. Lookahead address/result generation may be particularly beneficial for embodiments employing the x86 instruction set. Because of the nature the x86 instruction set, many of the instructions in a typical code sequence are versions of simple moves. One reason for this feature is that x86 instructions include two operands, both of which are source operands and one of which is a destination operand. Therefore, one of the source operands of each instruction is overwritten with an execution result. Furthermore, the x86 instruction set specifies very few registers for storing register operands. Accordingly, many instructions are moves of operands to and from a stack maintained within memory. Still further, many the instruction dependencies are dependencies upon the ESP/EBP registers and yet many of the updates to these registers are increments and decrements of the previously stored values.

To accelerate the execution of these instructions, lookahead/collapse unit **24** generates lookahead copies of the ESP and EBP registers for each of instructions decoded during a clock cycle. Additionally, lookahead/collapse unit **24** accesses future file **26** for register operands selected by each instruction. For each register operand, future file **26** may be storing either an execution result or a tag identifying a reorder buffer result queue entry corresponding to the most recent instruction having that register as a destination operand.

In one embodiment, lookahead/collapse unit **24** attempts to perform an address calculation for each instruction which: (i) includes a memory operand; and (ii) register operands used to form the address of the memory operand are available from future file **26** or lookahead copies of ESP/EBP. Additionally, lookahead/collapse unit **24** attempts to perform a result calculation for each instruction which: (i) does not include a memory operand; (ii) specifies an add/subtract operation (including increment and decrement); and (iii) register operands are available from future file **26** or lookahead copies of ESP/EBP. In this manner, many simple operations may be completed prior to instructions being sent to instruction windows **30A–30B**.

Lookahead/collapse unit **24** detects dependencies between a group of instructions being dispatched and collapses any execution results generated therein into instructions dependent upon those instruction results. Additionally, lookahead/collapse unit **24** updates future file **26** with the lookahead execution results. Instruction operations which are completed by lookahead/collapse unit **24** (i.e. address generations and/or instruction results are generated and load/store unit **36** or future file **26** and the result queue are updated) are not dispatched to instruction windows **30A–30B**.

Lookahead/collapse unit **24** allocates a result queue entry in reorder buffer/register file **28** for each instruction dispatched. In one particular embodiment, reorder buffer/register file **28** includes a result queue organized in a line-oriented fashion in which storage locations for execution results are allocated and deallocated in lines having enough storage for execution results corresponding to a maximum number of concurrently dispatchable instructions. If less than the maximum number of instructions are dispatched, then certain storage locations within the line are empty. Subsequently dispatched instructions use the next available line, leaving the certain storage locations empty. In one embodiment, the result queue includes 40 lines, each of which may store up to six execution results corresponding to concurrently dispatched instructions. Execution results are retired from the result queue in order into the register file included within reorder buffer/register file **28**. Additionally, the reorder buffer handles branch mispredictions, transmitting the corrected fetch address generated by the execution of the branch instruction to fetch/scan unit **18**. Similarly, instructions which generate other exceptions are handled within the reorder buffer. Results corresponding to instructions subsequent to the exception-generating instruction are discarded by the reorder buffer. The register file comprises a storage location for each architected register. For example, the x86 instruction set defines **8** architected registers. The register file for such an embodiment includes eight storage locations. The register file may further include storage locations used as temporary registers by a microcode unit in embodiments employing microcode units. Further details of one exemplary embodiment of future file **26** and reorder buffer/register file **28** are illustrated in FIG. **14** below.

Future file **26** maintains the speculative state of each architected register as instructions are dispatched by lookahead/collapse unit **24**. As an instruction having a register destination operand is decoded by lookahead/collapse unit **24**, the tag identifying the storage location within the result queue portion of reorder buffer/register file **28** assigned to the instruction is stored into the future file **26** storage location corresponding to that register. When the corresponding execution result is provided, the execution result is stored into the corresponding storage location (assuming that a subsequent instruction which updates the register has not been dispatched).

It is noted that, in one embodiment, a group of up to six instructions is selected from instruction queue **20** and moves through the pipeline within lookahead/collapse unit **24** as a unit. If one or more instructions within the group generates a stall condition, the entire group stalls. An exception to this rule is if lookahead/collapse unit **24** generates a split line condition due to the number of ESP updates within the group. Such a group of instructions is referred to as a “line” of instructions herein.

Instruction windows **30** receive instructions from lookahead/collapse unit **24**. Instruction windows **30** store the instructions until the operands corresponding to the instructions are received, and then select the instructions for execution. Once the address operands of an instruction including a memory operation have been received, the instruction is transmitted to one of the address generation units **34**. Address generation units **34** generate an address from the address operands and forward the address to load/store unit **36**. On the other hand, once the execution operands of an instruction have been received, the instruction is transmitted to one of the functional units **32** for execution. In one embodiment, each integer window **30A–30B** includes **25** storage locations for instructions. Each integer window **30A–30B** is configured to select up to two address generations and two functional unit operations for execution each clock cycle in the address generation units **34** and functional units **32** connected thereto. In one embodiment, instructions fetched from L0 I-cache **16** remain in the order fetched until stored into one of instruction windows **30**, at which point the instructions may be executed out of order.

In embodiments of processor **10** employing the x86 instruction set, an instruction may include implicit memory operations for load/store unit **36** as well as explicit functional operations for functional units **32**. Instructions having no memory operand do not include any memory operations, and are handled by functional units **32**. Instructions having a source memory operand and a register destination operand include an implicit load memory operation handled by load/store unit **36** and an explicit functional operation handled by functional units **32**. Instructions having a memory source/destination operand include implicit load and store memory operations handled by load/store unit **36** and an explicit functional operation handled by functional units **32**. Finally, instructions which do not have an explicit functional operation are handled by load/store unit **36**. Each memory operation results in an address generation handled either by lookahead/collapse unit **24** or address generation units **34**. Memory operations and instructions (i.e. functional operations) may be referred to herein separately, but may be sourced from a single instruction.

Address generation units **34** are configured to perform address generation operations, thereby generating addresses for memory operations in load/store unit **36**. The generated addresses are forwarded to load/store unit **36** via result buses **48**. Functional units **32** are configured to perform integer arithmetic/logical operations and execute branch instructions. Execution results are forwarded to future file **26**, reorder buffer/register file **28**, and instruction windows **30A–30B** via result buses **48**. Address generation units **34** and functional units **32** convey the result queue tag assigned to the instruction being executed upon result buses **48** to identify the instruction being executed. In this manner, future file **26**, reorder buffer/register file **28**, instruction windows **30A–30B**, and load/store unit **36** may identify execution results with the corresponding instruction. FPU/multimedia unit **40** is configured to execute floating point and multimedia instructions.

Load/store unit **36** is configured to interface with L1 D-cache **38** to perform memory operations. A memory operation is a transfer of data between processor **10** and an external memory. The memory operation may be an explicit instruction, or may be implicit portion of an instruction which also includes operations to be executed by functional units **32**. Load memory operations specify a transfer of data from external memory to processor **10**, and store memory operations specify a transfer of data from processor **10** to external memory. If a hit is detected for a memory operation within L1 D-cache **38**, the memory operation is completed therein without access to external memory. Load/store unit **36** may receive addresses for memory operations from lookahead/collapse unit **24** (via lookahead address calculation) or from address generation units **34**. In one embodiment, load/store unit **36** is configured perform up to three memory operations per clock cycle to L1 D-cache **38**. For this embodiment, load/store unit **36** may be configured to buffer up to 30 load/store memory operations which have not yet accessed D-cache **38**. The embodiment may further be configured to include a 96 entry miss buffer for buffering load memory operations which miss D-cache **38** and a 32 entry store data buffer. Load/store unit **36** is configured to perform memory dependency checking between load and store memory operations.

L1 D-cache **38** is a high speed cache memory for storing data. Any suitable configuration may be used for L1 D-cache **38**, including set associative and direct mapped configurations. In one particular embodiment, L1 D-cache **38** is a 128 KB two way set associative cache employing 64 byte lines. L1 D-cache **38** may be organized as, for example, 32 banks of cache memory per way. Additionally, L1 D-cache **38** may be a linearly addressed/physically tagged cache employing a TLB similar to L1 I-cache **14**.

External interface unit **42** is configured to transfer cache lines of instruction bytes and data bytes into processor **10** in response to cache misses. Instruction cache lines are routed to predecode unit **12**, and data cache lines are routed to L1 D-cache **38**. Additionally, external interface unit **42** is configured to transfer cache lines discarded by L1 D-cache **38** to memory if the discarded cache lines have been modified to processor **10**. As shown in FIG. **1**, external interface unit

42 is configured to interface to an external L2 cache via L2 interface **44** as well as to interface to a computer system via bus interface **46**. In one embodiment, bus interface unit **46** comprises an EV/6 bus interface.

Turning now to FIG. **2**, a block diagram of one embodiment of fetch/scan unit **18** is shown. Other embodiments are possible and contemplated. As shown in FIG. **2**, fetch/scan unit **18** includes a prefetch control unit **50**, a plurality of select next blocks **52A–52C**, an instruction select multiplexor (mux) **54**, an instruction scanner **56**, a branch scanner **58**, a branch history table **60**, a branch select mux **62**, a return stack **64**, an indirect address cache **66**, and a forward collapse unit **68**. Prefetch control unit **50** is coupled to L1 I-cache **14**, L0 I-cache **16**, indirect address cache **66**, return stack **64**, branch history table **60**, branch scanner **58**, and instruction select mux **54**. Select next block **52A** is coupled to L1 I-cache **14**, while select next blocks **52B–52C** are coupled to L0 I-cache **16**. Each select next block **52** is coupled to instruction select mux **54**, which is further coupled to branch scanner **58** and instruction scanner **56**. Instruction scanner **56** is coupled to instruction queue **20**. Branch scanner **58** is coupled to branch history table **60**, return stack **64**, and branch select mux **62**. Branch select mux **62** is coupled to indirect address cache **66**. Branch history table **60** and branch scanner **58** are coupled to forward collapse unit **68**, which is coupled to instruction queue **20**.

Prefetch control unit **50** receives branch prediction information (including target addresses and taken/not taken predictions) from branch scanner **58**, branch history table **60**, return stack **64**, and indirect address cache **66**. Responsive to the branch prediction information, prefetch control unit **50** generates fetch addresses for L0 I-cache **16** and a prefetch address for L1 I-cache **14**. In one embodiment, prefetch control unit **50** generates two fetch addresses for L0 I-cache **16**. The first fetch address is selected as the target address corresponding to the first branch instruction identified by branch scanner **58** (if any). The second fetch address is the sequential address to the fetch address selected in the previous clock cycle (i.e. the fetch address corresponding to the run selected by instruction select mux **54**).

L0 I-cache **14** provides the cache lines (and predecode information) corresponding to the two fetch addresses, as well as the cache lines (and predecode information) which are sequential to each of those cache lines, to select next blocks **52B–52C**. More particularly, select next block **52B** receives the sequential cache line corresponding to the sequential address and the next incremental cache line to the sequential cache line. Select next block **52C** receives the target cache line corresponding to the target address as well as the cache line sequential to the target cache line. Additionally, select next blocks **52B–52C** receive the offset portion of the corresponding fetch address. Select next blocks **52B–52C** each select a run of instruction bytes (and corresponding predecode information) from the received cache lines, beginning with the run section including the offset portion of the corresponding fetch address. Since the offset portion of each fetch address can begin anywhere within the cache line, the selected run may included portions of the fetched cache line and the sequential cache line to the fetched cache line. Hence, both the fetched cache line and the sequential cache line are received by select next blocks **52B–52C**.

Similarly, select next block **52A** receives a prefetched cache line (and corresponding predecode information) from L1 I-cache **14** and selects an instruction run therefrom. Since one cache line is prefetched from L1 I-cache **14**, the run

selected therefrom may comprise less than a full run if the offset portion of the prefetch address is near the end of the cache line. It is noted that the fetch cache lines from L0 I-cache **16** may be provided in the same clock cycle as the corresponding addresses are generated by prefetch control unit **50**, but the prefetch cache line may be a clock cycle delayed due to the larger size and slower access time of L1 I-cache **14**. In addition to providing the prefetched cache line to select next block **52A**, L1 I-cache **14** provides the prefetched cache line to L0 I-cache **16**. If the prefetched cache line is already stored within L0 I-cache **16**, L0 I-cache **16** may discard the prefetched cache line. However, if the prefetched cache line is not already stored in L0 I-cache **14**, the prefetched cache line is stored into L0 I-cache **16**. In this manner, cache lines which may be accessed presently are brought into L0 I-cache **16** for rapid access therefrom. According to one exemplary embodiment, L0 I-cache **16** comprises a fully associative cache structure of eight entries. A fully associative structure may be employed due to the relatively small number of cache lines included in L0 I-cache **16**. Other embodiments may employ other organizations (e.g. set associative or direct-mapped).

Prefetch control unit **50** selects the instruction run provided by one of select next blocks **52** in response to branch prediction information by controlling instruction select mux **54**. As will be explained in more detail below, prefetch control unit **50** receives target addresses from branch scanner **58**, return stack **64**, and indirect address cache **66** early in the clock cycle as well as at least a portion of the opcode byte of the first branch instruction identified by branch scanner **58**. Prefetch control unit **50** decodes the portion of the opcode byte to select the target address to be fetched from L0 I-cache **16** from the various target address sources and provides the selected target address to L0 I-cache **16**. In parallel, the sequential address to the fetch address selected in the previous clock cycle (either the target address or the sequential address from the previous clock cycle, depending upon the branch prediction from the previous clock cycle) is calculated and provided to L0 I-cache **16**. Branch prediction information (i.e. taken or not taken) is provided by branch history table **60** late in the clock cycle. If the branch instruction corresponding to the target address fetched from L0 I-cache **16** is predicted taken, then prefetch control unit **50** selects the instruction run provided by select next block **52C**. On the other hand, if the branch instruction is predicted not taken, then the instruction run selected by select next block **52B** is selected. The instruction run provided by select next block **52A** is selected if a predicted fetch address missed L0 I-cache **16** in a previous clock cycle and was fetched from L1 I-cache **14**. Additionally, the instruction run from L1 I-cache **14** is selected if the instruction run was prefetched responsive to a branch instruction have a 32 bit displacement or indirect target address generation or an L0 I-cache miss was fetched.

The selected instruction run is provided to instruction scanner **56** and branch scanner **58**. Instruction scanner **56** scans the predecode information corresponding to the selected instruction run to identify instructions within the instruction run. More particularly in one embodiment, instruction scanner **56** scans the start bits corresponding to each run section in parallel and identifies up to five instructions within each run section. Pointers to the identified instructions (offsets within the run section) are generated. The pointers, instruction bytes, and addresses (one per run section) are conveyed by instruction scanner **56** to instruction queue **20**. If a particular run section includes more than five instructions, the information corresponding to run sections subsequent to the particular run section is invalidated and the particular run section and subsequent run sections are rescanned during the next clock cycle.

Branch scanner **58** scans the instruction run in parallel with instruction scanner **56**. Branch scanner **58** scans the start bits and control transfer bits of the instruction run to identify the first two branch instructions within the instruction run. As described above, a branch instruction is identified by the control transfer bit corresponding to the start byte of an instruction (as identified by the start bit) being set. Upon locating the first two branch instructions, branch scanner **58** assumes that the instructions are relative branch instructions and selects the corresponding encoded target addresses from the instruction bytes following the start byte of the branch instruction. For embodiments employing the x86 instruction set, a nine bit target address (the displacement byte as well as the corresponding control transfer bit) is selected, and a 32 bit target address is selected as well. Furthermore, at least a portion of the opcode byte identified by the start and control transfer bits is selected. The target addresses and opcode bytes are routed to prefetch control unit **50** for use in selecting a target address for fetching from L0 I-cache **16**. The fetch addresses of each branch instruction (determined from the fetch address of the run section including each branch instruction and the position of the branch instruction within the section) are routed to branch history table **60** for selecting a taken/not-taken prediction corresponding to each branch instruction. Furthermore, the fetch addresses corresponding to each branch instruction are routed to branch select mux **62**, which is further routed to indirect address cache **66**. The target address of each branch instruction is routed to forward collapse unit **68**. According to one embodiment, branch scanner **58** is configured to scan each run section in parallel for the first two branch instructions and then to combine the scan results to select the first two branch instructions within the run.

Branch scanner **58** may further be configured to determine if a subroutine call instruction is scanned during a clock cycle. Branch scanner **58** may forward the fetch address of the next instruction following the detected subroutine call instruction to return stack **64** for storage therein.

In one embodiment, if there are more than two branch instructions within a run, the run is scanned again during a subsequent clock cycle to identify the subsequent branch instruction.

The fetch addresses of the identified branch instructions are provided to branch history table **60** to determine a taken/not taken prediction for each instruction. Branch history table **60** comprises a plurality of taken/not-taken predictors corresponding to the previously detected behavior of branch instructions. One of the predictors is selected by maintaining a history of the most recent predictions and exclusive ORing those most recent predictions with a portion of the fetch addresses corresponding to the branch instructions. The least recent (oldest) prediction is exclusive ORed with the most significant bit within the portion of the fetch address, and so forth through the most recent prediction being exclusive ORed with the least significant bit within the portion of the fetch address. Since two predictors are selected per clock cycle, the predictor corresponding to the second branch instruction is dependent upon the prediction of the first branch instruction (for exclusive ORing with the least significant bit of the corresponding fetch address). Branch history table **60** provides the second predictor by selecting both of the predictors which might be selected (i.e. the predictor that would be selected if the first branch instruction is predicted not-taken and the predictor that would be selected if the first branch instruction is predicted taken) and then selecting one of the two predictors based on the actual prediction selected for the first branch instruction.

Branch history table **60** receives information regarding the execution of branch instructions from functional units **32A–32D**. The history of recent predictions corresponding to the executed branch instruction as well as the fetch address of the executed branch instruction are provided for selecting a predictor to update, as well as the taken/not taken result of the executed branch instruction. Branch history table **60** selects the corresponding predictor and updates the predictor based on the taken/not taken result. In one embodiment, the branch history table stores a bimodal counter. The bimodal counter is a saturating counter which saturates at a minimum and maximum value (i.e. subsequent decrements of the minimum value and increments the maximum value cause no change in the counter). Each time a branch instruction is taken, the corresponding counter is incremented and each time a branch instruction is not taken, the corresponding counter is decremented. The most significant bit of the counter indicates the taken/not taken prediction (e.g. taken if set, not taken if clear). In one embodiment, branch history table **60** stores 64K predictors and maintains a history of the 16 most recent predictions. Each clock cycle, the predictions selected during the clock cycle are shifted into the history and the oldest predictions are shifted out of the history.

Return stack **64** is used to store the return addresses corresponding to detected subroutine call instructions. Return stack **64** receives the fetch address of a subroutine call instruction from branch scanner **58**. The address of the byte following the call instruction (calculated from the fetch address provided to return stack **64**) is placed at the top of return stack **64**. Return stack **64** provides the address stored at the top of the return stack to prefetch control unit **50** for selection as a target address if a return instruction is detected by branch scanner **58** and prefetch control unit **50**. In this manner, each return instruction receives as a target address the address corresponding to the most recently detected call instruction. Generally in the x86 instruction set, a call instruction is a control transfer instruction which specifies that the sequential address to the call instruction be placed on the stack defined by the x86 architecture. A return instruction is an instruction which selects the target address from the top of the stack. Generally, call and return instructions are used to enter and exit subroutines within a code sequence (respectively). By placing addresses corresponding to call instructions in return stack **64** and using the address at the top of return stack **64** as the target address of return instructions, the target address of the return instruction may be correctly predicted. In one embodiment, return stack **64** may comprise 16 entries.

Indirect address cache **66** stores target addresses corresponding to previous executions of indirect branch instructions. The fetch address corresponding to an indirect branch instruction and the target address corresponding to execution of the indirect branch instruction are provided by functional units **32A–32D** to indirect address cache **66**. Indirect address cache **66** stores the target addresses indexed by the corresponding fetch addresses. Indirect address cache **66** receives the fetch address selected by branch select mux **62** (responsive to detection of an indirect branch instruction) and, if the fetch address is a hit in indirect address cache **66**, provides the corresponding target address to prefetch control unit **50**. In one embodiment, indirect address cache **66** may comprise 32 entries.

According to one contemplated embodiment, if indirect address cache **66** detects a miss for a fetch address, indirect address cache **66** may be configured to select a target address to provide from one of the entries. In this manner, a "guess" at a branch target is provided in case an indirect branch instruction is decoded. Fetching from the guess may be performed rather than awaiting the address via execution of the indirect branch instruction. Alternatively, another contemplated embodiment awaits the address provided via execution of the indirect branch instruction.

According to one embodiment, prefetch control unit **50** selects the target address for fetching from L0 I-cache **16** from: (i) the first encoded target address corresponding to the first branch instruction identified by branch scanner **58**; (ii) the return stack address provided by return stack **64**; and (iii) a sequential address. Prefetch control unit **50** selects the first encoded target address if a decode of the opcode corresponding to the first instruction indicates that the instruction may be a relative branch instruction. If the decode indicates that the instruction may be a return instruction, then the return stack address is selected. Otherwise, the sequential address is selected. Indirect target addresses and 32 bit relative target addresses are prefetched from L1 I-cache **14**. Since these types of target addresses are often used when the target address is not near the branch instruction within memory, these types of target addresses are less likely to hit in L0 I-cache **16**. Additionally, if the second branch instruction is predicted taken and the first branch instruction is predicted not taken or the first branch instruction is a forward branch which does not eliminate the second branch instruction in the instruction run, the second target address corresponding to the second branch prediction may be used as the target fetch address during the succeeding clock cycle according to one embodiment.

It is noted that, if an encoded target address is selected, the actual target address may be presented to L0 I-cache **16**. Prefetch control unit **50** may be configured to precalculate each of the possible above/below target addresses and select the correct address based on the encoded target address. Alternatively, prefetch control unit **50** may record which L0 I-cache storage locations are storing the above and below cache lines, and select the storage locations directly without a tag compare.

Forward collapse unit **68** receives the target addresses and positions within the instruction run of each selected branch instruction as well as the taken/not taken predictions. Forward collapse unit **68** determines which instructions within the run should be cancelled based upon the received predictions. If the first branch instruction is predicted taken and is backward (i.e. the displacement is negative), all instructions subsequent to the first branch instruction are cancelled. If the first branch instruction is predicted taken and is forward but the displacement is small (e.g. within the instruction run), the instructions which are between the first branch instruction and the target address are cancelled. The second branch instruction, if still within the run according to the first branch instruction's prediction, is treated similarly. Cancel indications for the instructions within the run are set to instruction queue **20**.

Prefetch control unit **50** may be further configured to select a cache line within L0 I-cache **16** for replacement by a cache line provided from L1 I-cache **14**. In one embodiment, prefetch control unit **50** may use a least recently used (LRU) replacement algorithm.

Turning now to FIG. **3**, a block diagram of one embodiment of lookahead/collapse unit **24** is shown. Other embodiments are possible and contemplated. As shown in FIG. **3**, lookahead/collapse unit **24** includes a plurality of decode units **70A–70F**, an ESP/EBP lookahead unit **72**, a lookahead address/result calculation unit **74**, a dispatch control unit **76**, and an operand collapse unit **78**. Decode units **70A–70F** are coupled to receive instructions from alignment unit **22**. Decode units **70A–70F** are coupled to provide decoded instructions to FPU/multimedia unit **40**, ESP/EBP lookahead unit **72**, future file **26**, and lookahead address/result calculation unit **74**. ESP/EBP lookahead unit **72** is coupled to lookahead address/result calculation unit **74**, as is future file **26**. Lookahead address/result calculation unit **74** is further coupled load/store unit **36** and dispatch control unit **76**. Dispatch unit **76** is further coupled to operand collapse unit **78**, future file **26**, load/store unit **36**, and reorder buffer **28**. Operand collapse unit **78** is coupled to instruction windows **30**.

Each decode unit **70A–70F** forms an issue position to which alignment unit **22** aligns an instruction. While not indicated specifically throughout FIG. **3** for simplicity the drawing, a particular instruction remains within its issue position as the instruction moves through lookahead/collapse unit **24** and is routed to one of instruction windows **30A–30B** if not completed within lookahead/collapse unit **24**.

Decode units **70A–70F** route FPU/multimedia instructions to FPU/multimedia unit **40**. However, if the FPU/multimedia instructions include memory operands, memory operations are also dispatched to load/store unit **36** in response to the instruction through lookahead address/result calculation unit **74**. Additionally, if the address for the memory operations cannot be generated by lookahead address/result calculation unit **74**, an address generation operation is dispatch to one of address generation units **34A–34D** via instruction windows **30A–30B**. Still further, entries within reorder buffer **28** are allocated to the FPU/multimedia instructions for maintenance of program order. Generally, entries within reorder buffer **28** are allocated from decode units **70A–70F** for each instruction received therein.

Each of decode units **70A–70F** are further configured to determine: (i) whether or not the instruction uses the ESP or EBP registers as a source operand; and (ii) whether not the instruction modifies the ESP/EBP registers (i.e. has the ESP or EBP registers as a destination operand). Indications of these determinations are provided by decode units **70A–70F** to ESP/EBP lookahead unit **72**. ESP/EBP lookahead unit **72** generates lookahead information for each instruction which uses the ESP or EBP registers as a source operand. The lookahead information may include a constant to be added to the current lookahead value of the corresponding register and an indication of a dependency upon an instruction in a prior issue position. In one embodiment, ESP/EBP lookahead unit **72** is configured to provide lookahead information as long as the set of concurrently decoded instructions provided by decode units **70A–70F** do not include more than: (i) two push operations (which decrement the ESP register by a constant value); (ii) two pop operations (which increment ESP register by a constant value); (iii) one move to ESP register; (iv) one arithmetic/logical instruction having the ESP as a destination; or (v) three instructions which update ESP. If one of these restrictions is exceeded, ESP/EBP lookahead unit **72** is configured to stall instructions beyond those which do not exceed restrictions until the succeeding clock cycle (a "split line" case). For those instructions preceded, in the same clock cycle but in earlier issue positions, by instructions which increment or decrement the ESP register, ESP/EBP lookahead unit **72** generates a constant indicating the combined total modification to the ESP register of the preceding instructions. For those instructions preceded by a move or arithmetic operation upon the ESP or EBP registers, ESP/EBP lookahead unit **72** generates a value identifying the issue position containing the move or arithmetic instruction.

The lookahead values may be used by lookahead address/result calculation unit **74** to generate either a lookahead address corresponding to the instruction within the issue position (thereby inhibiting an address generation operation which would otherwise be performed by one of address generation units **34A–34D**) or a lookahead result corresponding to the instruction (thereby providing lookahead state to future file **26** earlier in the pipeline). Performance may be increased by removing address generation operations and/or providing lookahead state prior to functional units **32A–32D** and address generation units **34A–34D**. Many x86 code sequences include a large number of relatively simple operations such as moves of values from a source to destination without arithmetic/logical operation or simple arithmetic operations such as add/subtract by small constant or increment/decrement of a register operand. Accordingly, functional units **32A–32D** may typically execute the more complex arithmetic/logical operations and branch instructions and address generation units **34A–34D** may typically perform the more complex address generations. Instruction throughput may thereby be increased.

Decode units **70A–70F** are still further configured to identify immediate data fields from the instructions decoded therein. The immediate data is routed to lookahead address/result calculation unit **74** by decode units **70A–70F**. Additionally, decode unit **70A–70F** are configured to identify register operands used by the instructions and to route register operand requests to future file **26**. Future file **26** returns corresponding speculative register values or result queue tags for each register operand. Decode units **70** further provide dependency checking between the. line of instructions to ensure that an instruction which uses a result of an instruction within a different issue position receives a tag corresponding to that issue position.

Lookahead address/result calculation unit **74** receives the lookahead values from ESP/EBP lookahead units **72**, the immediate data from decode units **70A–70F**, and the speculative register values or result queue tags from future file **26**. Lookahead address/result calculation unit **74** attempts to generate either a lookahead address corresponding to a memory operand of the instruction, or a lookahead result if the instruction does not include a memory operand. For example, simple move operations can be completed (with respect to functional units **32** and address generation units **34**) if an address generation can be performed by lookahead address/result calculation unit **74**. In one embodiment, lookahead address/result calculation unit **74** is configured to compute addresses using displacement only, register plus displacement; ESP/EBP plus displacement, and scale-index-base addressing mode except for index or base registers being ESP/EBP. Load/store unit **36** performs the memory operation and returns the memory operation results via result buses **48**. Even if no address is generated for a memory operation by lookahead address/result calculation unit **74**, lookahead address/result calculation unit **74** indicates the memory operation and corresponding result queue tag to load/store unit **36** to allocate storage within load/store unit **36** for the memory operation.

Simple arithmetic operations which increment or decrement a source operand, add/subtract a small immediate value to a source operand, or add/subtract two register source operands may also be completed via lookahead address/result calculation unit **74** if the source operands are

available from future file **26** (i.e. a speculative register value is received instead of a result queue tag). Instructions completed by lookahead address/result calculation units **74** are indicated as completed and are allocated entries in reorder buffer **28** but are not dispatched to instruction windows **30**. Lookahead address/result calculation unit **74** may comprise, for example, an adder for each issue position along with corresponding control logic for selecting among the lookahead values, immediate data, and speculative register values. It is noted that simple arithmetic operations may still be forwarded to instruction windows **30** for generation of condition flags, according to the present embodiment. However, generating the functional result in lookahead address/result calculation unit **74** provides the lookahead state early, allowing subsequent address generations/instructions to be performed early as well.

Lookahead address/result calculation unit **74** may be configured to keep separate lookahead copies of the ESP/EBP registers in addition to the future file copies. However, if updates to the ESP/EBP are detected which cannot be calculated by lookahead address/result calculation unit **74**, subsequent instructions may be stalled until a new lookahead copy of the ESP/EBP can be provided from future file **26** (after execution of the instruction which updates ESP/EBP in the undeterminable manner).

Dispatch control unit **76** determines whether or not a group of instructions are dispatched to provide pipeline flow control. Dispatch control unit **76** receives instruction counts from instruction windows **30** and load/store counts from load/store unit **36** and, assuming the maximum possible number of instructions are in flight in pipeline stages between dispatch control units **76** and instruction windows **30** and load/store unit **36**, determines whether or not space will be available for storing the instructions to be dispatched within instruction windows **30** and/or load/store unit **36** when the instructions arrive therein. If dispatch control unit **76** determines that insufficient space will be available in load/store unit **36** and either instruction window **30**, dispatch is stalled until the instruction counts received by dispatch control unit **76** decrease to a sufficiently low value.

Upon releasing instructions for dispatch through dispatch control unit **76**, future file **26** and reorder buffer **28** are updated with speculatively generated lookahead results. In one embodiment, the number of non-ESP/EBP updates supported may be limited to, for example, two in order to limit the number of ports on future file **26**. Furthermore, operand collapse unit **78** collapses speculatively generated lookahead results into subsequent, concurrently decoded instructions which depend upon those results as indicated by the previously determined intraline dependencies. In this manner, the **25** dependent instructions receive the speculatively generated lookahead results since these results will not subsequently be forwarded from functional units **32A–32D**. Those instructions not completed by lookahead address/result calculation unit **74** are then transmitted to one of instruction windows **30A–30B** based upon the issue position to which those instructions were aligned by alignment unit **22**.

It is noted that certain embodiments of processor **10** may employ a microcode unit (not shown) for executing complex instructions by dispatching a plurality of simpler instructions referred to as a microcode routine. Decode units **70A–70F** may be configured to detect which instructions are microcode instructions and to route the microcode instructions to the microcode unit. For example, the absence of a directly decoded instruction output from a decode unit **70** which received a valid instruction may be an indication to the microcode unit to begin execution for the corresponding valid instruction. Is further noted that various storage devices are shown in FIGS. **2** and **3** (e.g. devices **79**A, **79**B, and similar devices in FIG. **2** and devices **79**C, **79**D and similar devices in FIG. **3**). The storage devices represent latches, registers, flip-flops and the like which may be used to separate pipeline stages. However, the particular pipeline stages shown in FIGS. **2** and **3** are but one embodiment of suitable pipeline stages for one embodiment of processor **10**. Other pipeline stages may be employed in other embodiments.

It is noted that, while the x86 instruction set and architecture has been used as an example above and may be used as an example below, any instruction set and architecture may be used. Additionally, displacements may be any desirable size (in addition to the 8 bit and 32 bit sizes used as examples herein). Furthermore, while cache line fetching may be described herein, it is noted that cache lines may be sectors, and sectors may be fetched, if desirable based upon cache line size and the number of bytes desired to be fetched.

Turning now to FIG. **4**, a block diagram of one embodiment of predecode unit **12** is shown. Other embodiments are possible and contemplated. As shown in FIG. **4**, predecode unit **12** includes an input instruction bytes register **80**, a fetch address register **82**, a byte predecoder **84**, a control unit **86**, a target generator **88**, a start and control transfer bits register **90**, an output instruction bytes register **92**, and a byte select mux **94**. Input instruction bytes register **80** is coupled to byte predecoder **84**, control unit **86**, target generator **88**, byte select mux **94**, and external interface unit **42**. Fetch address register **82** is coupled to L1 I-cache **14** and target generator **88**. Byte predecoder **84** is coupled to start and control transfer bits register **90** and control unit **86**. Control unit **86** is coupled to L1 I-cache **14**, byte select mux **94**, and target generator **88**. Target generator **88** is coupled to byte select mux **94**, which is further coupled to output instruction bytes register **92**. Output instruction bytes register **92** and start and control transfer bits register **90** are further coupled to L1 I-cache **14**.

Upon detection of an L1 I-cache miss, predecode unit **12** receives the linear fetch address corresponding to the miss into fetch address register **82**. In parallel, external interface unit **42** receives the corresponding physical fetch address and initiates an external fetch for the cache line identified by the fetch address. External interface unit **42** provides the received instruction bytes to input instruction bytes register **80**.

Byte predecoder **84** predecodes the received instruction bytes to generate corresponding start and control transfer predecode bits. The generated predecode information is stored into start and control transfer bits register **90**. Because instructions can have boundaries at any byte within the cache line due to the variable length nature of the x86 instruction set, byte predecoder **84** begins predecoding at the offset within the cache line specified by the fetch address stored within fetch address register **82**. The byte specified by the offset is assumed to be the first byte of an instruction (i.e. the corresponding start bit is set). Byte predecoder **84** predecodes each byte beginning with the first byte to determine the beginning of each instruction and to detect branch instructions. Branch instructions result in the control transfer bit corresponding to the start byte of the branch instruction being set by byte predecoder **84**. Additionally, byte predecoder **84** informs control unit **86** if the branch instruction is a relative branch instruction and indicates the position of the instruction subsequent to the branch instruction within the cache line. In one embodiment, byte predecoder **84** is configured to predecode four bytes per clock cycle in parallel.

Responsive to the signal from byte predecoder **84** indicating that a relative branch instruction has been detected, control unit **86** causes target generator **88** to generate the target address corresponding to the relative branch instruction. The displacement byte or bytes are selected from the instruction bytes stored in register **80**. Additionally, the fetch address stored in fetch address register **82** (with the offset portion replaced by the position of the instruction subsequent to the branch instruction) is provided to target generator **88**. Target generator **88** adds the received address and displacement byte or bytes, thereby generating the target address. The generated target address is then encoded for storage as a replacement for the displacement field of the relative branch instruction. Additionally, control unit **86** select the output of target generator **88** to be stored into output instruction bytes register **92** instead of the corresponding displacement bytes of the relative branch instruction from input instruction bytes register **80**. Other instruction bytes are selected from input instruction bytes register **80** for storage in output instruction bytes register **92** as those bytes are predecoded by byte predecoder **84**. Once byte predecoder **84** has completed predecode of the cache line and each relative branch instruction has had its displacement replaced by an encoding of the target address, control unit **86** asserts a predecode complete signal to L1 I-cache **14**, which then stores the output instruction bytes and corresponding start and control transfer bits.

As described above, for relative branch instructions having small displacement fields (e.g. a single displacement byte) the control transfer bit corresponding to the displacement byte is used in addition to the displacement byte to store the encoding of the target address. Target generator **88** signals byte predecoder **84** with the appropriate control transfer bit, which byte predecoder **84** stores in the corresponding position within start and control transfer bits register **90**.

It is noted that, if a relative branch instruction spans the boundary between two cache lines (i.e. a first cache line stores a first portion of the instruction and the succeeding cache line stored the remaining portion), predecode unit **12** may be configured to fetch the succeeding cache line in order to complete the predecoding for the relative branch instruction. It is further noted that predecode unit **12** may be configured to handle multiple outstanding cache lines simultaneously.

Turning next to FIG. **4**A, a block diagram of one embodiment of target generator **88** is shown. Other embodiments are possible and contemplated. As shown in FIG. **4**A, target generator **88** includes a displacement mux **100**, a sign extend block **102**, an adder **104**, and a displacement encoder **106**. Displacement mux **100** is coupled to input instruction bytes register **80** and sign extend block **102**, and receives control signals from control unit **86**. Sign extend block **102** is coupled to an input of adder **104** and receives control signals from control unit **86**. The second input of adder **104** is coupled to receive the fetch address from fetch address register **82** (except for the offset bits) concatenated with a position within the cache line from control unit **86**. Adder **104** is further coupled to displacement encoder **106** which receives control signals from control unit **86**. Displacement encoder **106** is further coupled to byte select mux **94** and byte predecoder **84**.

Displacement mux **100** is used to select a displacement byte or bytes from the relative branch instruction. In the present embodiment, displacements may be one or four bytes. Accordingly, displacement mux **100** selects four bytes from input instruction bytes register **80**. If a one byte displacement is included in the relative branch instruction, the displacement is selected into the least significant of the four bytes. The remaining three bytes may be zeros or may be prior bytes within input instruction bytes register **80**. Sign extend block **102**, under control from control unit **86**, sign extends the one byte displacement to a four byte value. On the other hand, a four byte displacement is selected by displacement mux **100** and is not modified by sign extend block **102**. It is noted that larger addresses may be employed by processor **10**. Generally, the displacement may be sign extended to the number of bits within the address.

Displacement encoder **106** receives the target address calculated by adder **104** and encodes the target address into a format storable into the displacement bytes. In the present embodiment, a four byte displacement stores the entirety of the target address. Hence, displacement encoder **106** passes the target address unmodified to byte select mux **94** for storage in output instruction bytes register **92**. Additionally, the control transfer bits corresponding to the displacement bytes are. not used. For one byte displacements, the target address is encoded. More particularly, a portion of the displacement byte is used to store the offset of the target address within the target cache line (e.g. in the present embodiment, 6 bits to store a 64 byte offset). The remaining portion of the displacement byte and the corresponding control transfer bit is encoded with a value indicating the target cache line as a number of cache lines above or below the cache line identified by the fetch address stored in fetch address register **82**. Accordingly, displacement encoder **106** is coupled to receive the fetch address from fetch address register **82**. Displacement encoder **106** compares the fetch address to the target address to determine not only the number of cache lines therebetween, but the direction. Upon generating the encoding, displacement encoder **106** transmits the modified displacement byte to byte select mux **94** for storage in output instruction bytes register **92** and also transmits the value for the control transfer bit corresponding to the displacement byte to byte predecoder **84**.

As an alternative to employing adder **104** to calculate target addresses for small displacement fields, displacement encoder **106** may directly generate the encoded target address (above below value and cache line offset) by examining the value of the displacement field and the position of the branch instruction within the cache line.

Turning now to FIG. **5**, a diagram illustrating an exemplary relative branch instruction **110** having an eight bit displacement according to the x86 instruction set is shown. Relative branch instruction **110** includes two bytes, an opcode byte **112** which is also the first byte of the instruction and a displacement byte **114**. Opcode byte **112** specifies that instruction **110** is a relative branch instruction and that the instruction has an eight bit displacement. Displacement byte **114** has been updated with an encoding of the target address. The encoding includes a cache line offset portion labeled "CL offset" (which comprises six bits in the current embodiment but may comprise any number bits suitable for the corresponding instruction cache line size) and a relative cache line portion labeled "LI2" in the control transfer bit corresponding to displacement byte **114** and "LI1 LI0" within displacement byte **114**.

FIG. **5** also illustrates the start and control transfer bits corresponding to instruction **110**. The start bit for each byte is labeled "S" in FIG. **5** with a box indicating the value of the bit, and the control transfer bit is labeled "C" with a box indicating the value of the bit. Accordingly, the start bit corresponding to opcode byte **112** is set to indicate that opcode byte **112** is the beginning of an instruction and the control transfer bit corresponding to opcode byte **112** is also set to indicate that the instruction beginning at opcode byte **112** is a control transfer instruction. The start bit corresponding to displacement byte **114**, on the other hand, is clear because displacement byte **114** is not the beginning of an instruction. The control transfer bit corresponding to displacement byte **114** is used to store a portion of the relative cache line portion of the encoded target address.

Turning next to FIG. **6**, an exemplary relative branch instruction **120** having a 32-bit displacement according to the x86 instruction set is shown. Instruction **120** includes an opcode field **122** comprising two bytes and a displacement field **124** comprising four bytes. Similar to FIG. **5**, FIG. **6** illustrates the start and control transfer bits for each byte within instruction **120**. Accordingly, two start bits and two control transfer bits are illustrated for opcode field **122**, and one start bit and control transfer bit are illustrated for each byte within displacement field **124**.

The first start bit corresponding to opcode field **122** (i.e. the start bit corresponding to the first byte of opcode field **122**) is set, indicating that the first byte of opcode field **122** is the beginning of an instruction. The first control transfer bit corresponding to opcode field **122** is also set indicating that instruction **120** is a control transfer instruction. The second start bit corresponding to opcode field **122** is clear, as the second byte within opcode field **122** is not the start of instruction. The control transfer bit corresponding to the second opcode byte is a don't care (indicated by an "x").

Since displacement field **124** is large enough to contain the entirety of the target address corresponding to instruction **120**, the control transfer bits corresponding to the displacement bytes are also don't cares. Each start bit corresponding to displacement byte is clear, indicating that that these bytes are not the start of an instruction.

Turning now to FIG. **7**, a diagram of an exemplary set of instructions **130** from the x86 instruction set are shown, further illustrating use of the start and control transfer bits according to one embodiment of processor **10**. Similar to FIGS. **5** and **6**, each byte within the set of instructions **130** is illustrated along with a corresponding start bit and control transfer bit.

The first instruction within set of instructions **130** is an add instruction which specifies addition of a one byte immediate field to the contents of the AL register and storing the result in the AL register. The add instruction is a two byte instruction in which the first byte is the opcode byte and the second byte is the one byte immediate field. Accordingly, the opcode byte is marked with a set start bit indicating the beginning of the instruction. The corresponding control transfer bit is clear indicating that the add instruction is not a branch instruction. The start bit corresponding to the immediate byte is clear because the immediate byte is not the start of an instruction, and the control transfer bit is a don't care.

Subsequent to the add instruction is a single byte instruction (an increment of the EAX register). The start bit corresponding to the instruction set because the byte is the beginning of instruction. The control transfer bit is clear since the increment is not a branch instruction.

Finally, a second add instruction specifying the addition of a one byte immediate field to the contents of the AL register is shown subsequent to the increment instruction. The start bit corresponding to the opcode of the add instruction is set, and the control transfer bit is clear. The increment instruction followed by the add instruction illustrates that consecutive bytes can have start bits which are set in the case where a single byte is both the start boundary and end boundary of the instruction.

Turning now to FIG. **8**, a block diagram of one embodiment of branch scanner **58** is shown for use with the x86 instruction set. Other embodiments are possible and contemplated. In the embodiment of FIG. **8**, branch scanner **58** includes a scan block **140**, section target muxes **142**A–**142**D, and run target muxes **144**A–**144**D. Scan block **140** is coupled to receive the start and control transfer bits corresponding to a run section from select next blocks **52** through instruction select mux **54**. Branch scanner **58** further includes additional scan blocks similar to scan block **140** for scanning the start and control transfer bits corresponding to the remaining run sections of the selected run. Scan block **140** is coupled to section target muxes **142**A–**142**D to provide selection controls thereto. Additionally, scan block **140** (and similar scan blocks for the other run sections) provide selection controls for run target muxes **144**A–**144**D. Each of section target muxes **142**A–**142**B is coupled to receive the instruction bytes corresponding to the run section scanned by scan block **140** as well as the corresponding control transfer bits. Each of section target muxes **142**C–**142**D are coupled receive the instruction bytes corresponding to the run section as well, but may not receive the corresponding control transfer bits. Each of section target muxes **142**A–**142**D is coupled to respective one of run target muxes **144**A–**144**D as shown in FIG. **8**. The outputs of run target muxes **144**A and **144**B are coupled to prefetch control unit **50** and to branch history table **60**. The outputs of run target muxes **144**C and **144**D are coupled to prefetch control unit **50**.

Scan block **140** is configured to scan the start and control transfer bits received therein in order to locate the first two branch instructions within the run section. If a first branch instruction is identified within the run section, scan block **140** directs section target mux **142**A to select the opcode byte, which is the byte for which both the start and control transfer bits are set, and the immediately succeeding byte and the control transfer bit corresponding to the immediately succeeding byte, which collectively form the encoded target address if the first branch instruction includes an eight bit relative displacement. Similarly, if a second branch instruction is identified within the run section, scan block **140** directs section target mux **142**B to select the opcode byte of the second branch instruction and the immediately succeeding byte and the control transfer bit corresponding to the immediately succeeding byte. In this manner, the opcode byte and target address corresponding to the first two relative branch instructions having eight bit displacement are selected. Additionally, the position of each branch instruction within the run section is identified by scan block **140**.

Scan block **140** is further configured to control section target mux **142**C in response to detecting the first branch instruction. More particularly, scan block **140** selects the four consecutive instruction bytes beginning with the second byte following the start byte of the first branch instruction (i.e. beginning with the byte two bytes subsequent to the start byte of the first branch instruction within the cache line). These consecutive instruction bytes are the encoded target address if the first branch instruction includes a 32-bit relative displacement. Similarly, scan block **140** controls section target mux **142**D to select the four consecutive start bytes beginning with the second byte following the start byte of the second branch instruction. In this manner, the target address corresponding to the first two relative branch instructions having 32-bit displacements are selected. Prefetch control unit **50** is configured to determine whether or not either: (i) the target address selected by section target mux **142**A; (ii) the target address selected by section target mux **142**C; or (iii) a target address from return stack **64** or indirect address cache **66** corresponds to the first branch instruction. Similarly, prefetch control unit **50** is configured determine whether or not either: (i) the target address selected by section target mux **142**B; (ii) the target address selected by section target mux **142**D; or (iii) a target address from return stack **64** or indirect address cache **66** corresponds to the second branch instruction.

Scan block **140**, in conjunction with similar scan blocks for the other sections of the run, controls run target muxes **144**A–**144**D to select target information corresponding to the first two branch instructions within the run. Accordingly, run target mux **144**A selects the target address (i.e. the immediately succeeding byte and corresponding control transfer bit), opcode, and position of the first branch instruction within the run. Similarly, run target mux **144**B selects the target address, opcode, and position of the second branch instruction within the run. Run target muxes **144**C–**144**D select 32-bit target addresses corresponding to the first and second branch instructions, respectively.

Turning next to FIG. **9**, a block diagram of one embodiment of prefetch control unit **50** is shown. Other embodiments are possible contemplated. As shown in FIG. **9**, prefetch control unit **50** includes a decoder **150**, a fetch address mux **152**, an incrementor **154**, and an L1 prefetch control unit **156**. Decoder **150** is coupled to receive the first branch opcode corresponding to the first branch instruction within the run from branch scanner **58** and to reorder buffer **28** to receive a misprediction redirection indication and corresponding corrected fetch address. Additionally, decoder **150** is coupled to fetch address mux **152** and L1 prefetch control unit **156**. Fetch address mux **152** is coupled to receive the first target address corresponding to the first branch instruction within the run as selected by run target mux **144**A. The second target address corresponding to the second branch instruction address is also provided to fetch address mux **152** with a one clock cycle delay. Additionally, fetch address mux **152** is configured to receive the return address provided by return stack **64**, the corrected fetch address provided by reorder buffer **28** upon misprediction redirection, and the sequential address to the address fetched in the previous clock cycle (generated by incrementor **154**). Fetch address mux **152** is coupled to provide the target fetch address to L0 I-cache **16** and to L1 prefetch control unit **156**. L1 prefetch control unit **156** is further coupled to L0 I-cache **16** to receive a miss indication, to indirect address cache **66** to receive a predicted indirect target address, to branch scanner **58** to receive 32-bit target addresses corresponding to relative branch instructions, to reorder buffer **28** to receive branch misprediction addresses, and to L1 I-cache **14** to provide an L1 prefetch address. Prefetch control unit **50** provides a sequential fetch address to L0 I-cache **16** via a register **158**.

Decoder **150** is configured to decode the opcode correspond to the first identified branch instruction from branch scanner **58** in order to select the target fetch address for L0 I-cache **16**. In order provide the target fetch address as rapidly is possible, decoder **150** decodes only a portion of the opcode byte received from branch scanner **58**. More particularly, for the x86 instruction set, decoder **150** may decode the four most significant bits of the opcode byte identified by the set start and control transfer bits to select one of the first target address from branch scanner **58**, the return address from return stack **64**, and the sequential address. FIG. **10**, described in more detail below, is a truth table corresponding to one embodiment of decoder **150**. Because only a subset of the bits of the opcode byte are decoded, fewer logic levels may be employed to generate the selection controls for fetch address mux **152**, thereby allowing rapid target address selection. If the target address selected responsive to the decode is incorrect, the fetched instructions may be discarded and the correct fetch address may be generated during a subsequent clock cycle.

Because the branch prediction corresponding to the first branch instruction within the run is not available until late in the clock cycle in which the fetch address is selected, decoder **150** does not attempt to select the second branch target address as the target fetch address. If the first branch instruction is predicted not taken, via branch history table **60**, the second target address corresponding to the second identified branch instruction (if any) may be fetched in a subsequent clock cycle if the second branch instruction is predicted taken by branch history table **60**. Also, if the first branch is predicted taken but the first target address is within the same run as the first branch, the sequential address is selected. If the first branch does not branch past the second branch within the run, the second target address is selected during the subsequent clock cycle. Similarly, if the first branch instruction uses an indirect target address or 32-bit relative target address, fetch address mux **152** may select an address and the fetched instructions may be discarded in favor of instructions at the actual branch target.

L1 prefetch control unit **156** generates an L1 prefetch address for L1 I-cache **14**. The cache line corresponding to the L1 prefetch address is conveyed to L0 I-cache **16** for storage. L1 prefetch control unit **156** selects the prefetch address from one of several sources. If a branch misprediction is signalled by reorder buffer **28**, the sequential address to the corrected fetch address provided by reorder buffer **28** is selected since the other address sources are based upon instructions within the mispredicted path. If no branch misprediction is signalled and an L0 fetch address miss is detected, L1 prefetch control unit **156** selects the L0 fetch address miss for prefetching. If no miss is detected, L1 prefetch control unit **156** selects either the indirect address provided by indirect address cache **66** or a 32-bit branch target address from branch scanner **58** responsive to signals from decoder **150**. If no signals are received from decoder **150**, L1 prefetch control unit **156** prefetches the cache line sequential to the target address selected by fetch address **152**.

Indirect addresses and 32-bit target addresses are not fetched from L0 I-cache **16** because these types of target addresses are typically selected by a programmer when the target instruction sequence is not spatially located within memory near the branch instruction. Because L0 I-cache **16** stores a small number of cache lines most recently accessed in response to the code sequence being executed, it may be statistically less likely that the target instruction sequence is stored in the L0 I-cache **16**.

Incrementor **154** is configured to increment the fetch address corresponding to the run selected for dispatch based on the branch prediction information received from branch history table. **60**. Prefetch control unit **50** includes logic (not shown) for selecting the run, via instruction select multiplexor **54**, based on L0 I-cache hit information as well as the branch prediction information. This logic also causes incrementor **154** to increment the fetch address corresponding to the selected run (either the sequential fetch address provided from register **158** or the target fetch address provided from fetch address mux **152**). Accordingly, the sequential fetch address for the subsequent clock cycle is generated and stored in register **158**.

Turning next to FIG. **10**, a truth table **160** corresponding to one embodiment of decoder **150** employed within one embodiment of processor **10** employing the x86 instruction set is shown. Other embodiments are possible and contemplated. As shown in FIG. **10**, opcodes having the four most significant bits equal to (in hexadecimal) 7, E, or 0 result in the first target address being selected by fetch address mux **152**. Opcodes having the four most significant bits equal to C result in the return address from return stack **64** being selected, and opcodes having the four most significant bits equal to F cause the sequential address to be selected.

Branch instruction opcodes having the four most significant bits equal to 7 are conditional jump instructions having eight bit relative displacements. Accordingly, an opcode corresponding to a set start bit and set control transfer bit which has the four most significant bits equal to 7 correctly selects the target address provided from run target mux **144A**. Branch instruction opcodes having the four most significant bits equal to E may be conditional jump instructions with eight bit relative displacements, or call or unconditional jump instructions having either eight bit relative displacements or 32 bit relative displacements. For these cases, decoder **150** selects the first target address provided by run target mux **144A** and, if further decode indicates that a 32-bit displacement field is included in the branch instruction, the instructions fetched in response to the selection are discarded and the correct fetch address is prefetch from L1 I-cache **14** via L1 prefetch control unit **156** receiving the 32-bit fetch address from branch scanner **58**. Finally, branch instruction opcodes having the four most significant bits equal to 0 specify 32-bit relative displacements. Since decoder **150** cannot select the 32 bit target address for fetching from L0 I-cache **16** in the present embodiment, decoder **150** selects the first target address provided from branch scanner **58** and signals L1 prefetch control unit **156** to select the 32-bit branch target address from branch scanner **58** for prefetching from L1 I-cache **14**.

Branch instruction opcodes having the four most significant bits equal to C are return instructions, and hence the return address provided by return address stack **64** provides the predicted fetch address. On the other hand, branch instruction opcodes having the four most significant bits equal to F are call or unconditional jump instructions which use indirect target address generation. The indirect address is not provided to fetch address mux **152**, and hence a default selection of the sequential address is performed. The instructions fetched in response to the sequential address are discarded and instructions prefetched from L1 I-cache **14** are provided during a subsequent clock cycle.

As truth table **160** illustrates, predecode of just a portion of the instruction byte identified by the start and control transfer bits may be used to select a target fetch address for L0 I-cache **16**. Accordingly, prefetch control unit **50** and branch scanner **58** may support high frequency, single cycle L0 I-cache access.

Turning next to FIG. **10A**, a flowchart is shown illustrating operation of one embodiment of decoder **150**. Other embodiments are possible and contemplated. While shown as a serial series of steps in FIG. **10A**, it is understood that the steps illustrated may be performed in any suitable order, and may be performed in parallel by combinatorial logic employed within decoder **150**.

Decoder **150** determines if a branch misprediction is being signalled by reorder buffer **28** (decision block **192**). If a misprediction is signalled, the corrected fetch address received from reorder buffer **28** is selected (step **193**). On the other hand, if a misprediction is not signalled, decoder **150** determines if the second target address corresponding to the second branch instruction identified during the previous clock cycle by branch scanner **58** is to be fetched (decision block **194**). The second target address may be fetched if the first branch instruction was predicted not-taken and the second branch instruction was predicted taken. Additionally, the second target address may be fetched if the first branch instruction was predicted taken, but was a small forward displacement which does not cancel the second branch instruction, and the second branch instruction was predicted taken. If the second target address is to be fetched, decoder **150** selects the second target address (which was received in the previous clock cycle and is one clock cycle delayed in reaching fetch address mux **152**—step **195**). Finally, if the second target address is not to be fetched, decoder **150** operates according to truth table **160** described above (step **196**).

Turning now to FIG. **11**, a flowchart is shown illustrating operation of one embodiment of L1 prefetch control unit **156**. Other embodiments are possible and contemplated. While shown as a serial series of steps in FIG. **11**, it is understood that the steps illustrated may be performed in any suitable order, and may be performed in parallel by combinatorial logic employed within L1 prefetch control unit **156**.

If a branch misprediction redirection is received by L1 prefetch control unit **156** (decision block **170**), the sequential cache line to the cache line corresponding to the corrected fetch address is prefetched from L1 I-cache **14** (step **172**). On the other hand, if a branch misprediction redirection is not received, L1 prefetch control unit **156** determines if an L0 I-cache miss has occurred (decision block **174**). If an L0 I-cache miss is detected, the address missing L0 I-cache **16** is prefetched from L1 I-cache **14** (step **176**). In the absence of an L0 I-cache miss, L1 prefetch control unit **156** determines if either an indirect target address or a 32-bit relative target address has been detected by decoder **150** (decision block **178**). If such a signal is received, the indirect address received from indirect address cache **66** or the 32-bit relative target address received from branch scanner **58** is prefetched from L1 I-cache **14** (step **180**). Finally, if no indirect target address or 32-bit relative target address is signalled, L1 prefetch control unit **156** prefetches the next sequential cache line to the current target fetch address (step **182**).

Turning now to FIG. **12**. a table **190** is shown illustrating the fetch results corresponding to one embodiment of processor **10** for various target addresses and branch predictions corresponding to the first and second branch instructions identified within an instruction run. Other embodiments are possible contemplated. As used in table **190**, a small forward target is a target which lies within the current run. Conversely, a large forward target is a target which does not lie within the current run. A target is forward if the target address is numerically greater than the address of the branch instruction, and backward if the target address is numerically smaller than the address of the branch instruction. The taken/not taken prediction is derived from branch history table **60**. As illustrated by the footnote, results corresponding to the second branch prediction may be delayed by a clock cycle according to one embodiment. Therefore, processor **10** may assume not taken for the second branch prediction (i.e. fetch the sequential address) and, if the second branch prediction indicates taken, the fetch may be corrected during the subsequent clock cycle.

The result column in table **190** lists several results. The term "squash" when used in the result column of table **190** indicates which instructions are deleted from instruction queue **20** via signals from forward collapse unit **68** shown in FIG. **2**. Additionally, the target or sequential address to be fetched responsive to the first and/or second branch instructions is indicated followed by parenthetical notation as to which of L0 I-cache **16** (L0 notation) or L1 I-cache **14** (L1 notation) the target or sequential address is conveyed.

Turning next to FIG. **13**, a block diagram of one exemplary embodiment of instruction queue **20** is shown. Other embodiments are possible and contemplated. In the embodiment of FIG. **13**, instruction queue **20** includes run storages **300A–300B**, scan data storages **302A–302B**, and address storages **304A–304B**. Additionally, instruction queue **20** includes a mux **306** and a control unit **308**. A run of instructions is provided to instruction queue **20** from fetch/scan unit **18** via a run bus **310**; corresponding scan data is provided on a scan data bus **312**; and corresponding addresses (one per run section) are provided on a run addresses bus **14**. Instruction queue **20** provides a set of selected instruction bytes to alignment unit **22** on instruction bytes bus **316**, pointers to instructions within the instruction bytes on an instruction pointers bus **318**, and addresses for the run sections comprising the set of selected instruction bytes on an addresses bus **320**. Run bus **310** is coupled to run storages **300A–300B**, while scan data bus **312** is coupled to scan data storages **302A–302B** and address storages **304A–304B** are coupled to run addresses bus **314**. Storages **300A–300B**, **302A–302B**, and **304A–304B** are coupled to mux **306**, which is further coupled to buses **316–320**. Control unit **308** is coupled to mux **306** and scan data storages **302A–302B**.

Fetch/scan unit **18**, and more particularly instruction scanner **56** according to the embodiment of FIG. **2**, provides a run of instructions and associated information to instruction queue **20** via buses **310–314**. Control unit **308** allocates one of run storages **300A–300B** for the instruction bytes comprising the instruction run, and a corresponding scan data storage **302A–302B** and address storage **304A–304B** for the associated information. The scan data includes instruction pointers which identify: (i) the start byte and end byte as offsets within a run section; as well as (ii) the run section within which the instruction resides. According to one particular embodiment, up to five instructions may be identified within an eight byte run section, and there are up to three run sections in a run for a total of up to **15** instructions pointers stored within a scan data storage **302**. Additionally, address storages **304** store an address corresponding to each run section.

Control unit **308** examines the instructions pointers within scan data storages **302A–302B** to identify instructions within a set of contiguous run sections for dispatch to alignment unit **22**. In one particular embodiment, up to six instructions are identified within up to four contiguous run sections. The run sections may be stored in one of run storages **300A** or **300B**, or some run sections may be selected from one of run storages **300A–300B** and the other run sections may be selected from the other one of run storages **300A–300B**. A first run section is contiguous to a second run section if the first run section is next, in speculative program order, to the second run section. It is noted that mux **306**, while illustrated as a single mux in FIG. **13** for simplicity in the drawing, may be implemented by any suitable parallel or cascaded set of multiplexors.

Control unit **308** provides a set of selection signals to mux **306** to select the set of run sections including the selected instructions, as well as the instruction pointers corresponding to the selected instructions. Additionally, the address for each selected run section is selected. The run sections are provided upon instruction bytes bus **316**, while the corresponding instruction pointers and addresses are provided upon instruction pointers bus **318** and addresses bus **320**, respectively.

Turning next to FIG. **14**, a block diagram of one embodiment of future file **26** and reorder buffer/register file **28** is shown in more detail. Other embodiments are possible and contemplated. In the embodiment of FIG. **14**, future file **26** is shown along with a register file **28**A and a reorder buffer **28**B. Future file **26** is coupled to register file **28**A, result buses **48**, a set of source operand address buses **330**, a set of source operand buses **332**, and a set of lookahead update buses **334**. Reorder buffer **28**B is coupled to register file **28**A, result buses **48**, and dispatched instructions buses **336**.

As instructions are decoded by decode units **70** within lookahead/collapse unit **24**, the register source operands of the instructions are routed to future file **26** via source operand address buses **330**. Future file **26** provides either the most current speculative value of each register, if the instruction generating the most current value has executed, or a reorder buffer tag identifying the instruction which will generate the most current value, upon source operands buses **332**. Additionally, one of the source operands may be indicated to be a destination operand. Future file **26** updates the location corresponding to the destination register with the reorder buffer tag to be assigned to the corresponding instruction in response to the destination operand.

Future file **26** additionally receives updates from lookahead/collapse unit **24**. Lookahead results generated by lookahead address/result calculation unit **74** are provided to future file **26** via lookahead update buses **334**. By providing lookahead updates from lookahead address/result calculation unit **74**, speculative execution results may be stored into future file **26** more rapidly and may thereby be available more rapidly to subsequently executing instructions. Subsequent instructions may thereby be more likely to achieve lookahead result calculation. In one embodiment, to. reduce the number of ports on future file **26**, the number of lookahead updates is limited (for example, 2 updates may be allowable). Since the ESP updates are already captured by lookahead/collapse unit **24**, those updates need not be stored into future file **26**. Furthermore, not every issue position will have a speculative update for future file **26**. Accordingly, fewer speculative updates, on average, may be needed in future file **26** and therefore limiting the number of updates may not reduce performance.

Instruction results are provided upon result buses **48**. Future file **26** receives the results and compares the corresponding reorder buffer tags (also provided upon result buses **48**) to the reorder buffer tags stored therein to determine whether or not the instruction result comprises the most recent speculative update to one of the architected registers. If the reorder buffer tag matches one of the reorder buffer tags in the future file, the result is capture by future file **26** and associated with the corresponding architected register.

Future file **26** is coupled to register file **28**A to receive a copy of the architected registers stored therein when an exception/branch misprediction is detected and retired. Reorder buffer **28**B may detect exceptions and branch mispredictions from the results provided upon result buses **48**, and may signal register file **28**A and future file **26** if a copy of the architected registers as retired in register file **28**A is to be copied to future file **26**. For example, upon retiring an instruction having an exception or branch misprediction, the copy may be performed. In this manner, future file **26** may be recovered from incorrect speculative execution.

Reorder buffer **28**B receives the dispatched instructions from lookahead/collapse unit **24** via dispatched instructions bus **336**. The dispatched instructions may be provided to reorder buffer **28**B upon a determination by dispatch control unit **76** that instructions are to be dispatched. Additionally, reorder buffer **28**B receives execution results upon results buses **48** and retires the results, in program order, to register file **28**A.

Turning now to FIG. **15**, a block diagram of one embodiment of a computer system **200** including processor **10** coupled to a variety of system components through a bus bridge **202** is shown. Other embodiments are possible and contemplated. In the depicted system, a main memory **204** is coupled to bus bridge **202** through a memory bus **206**, and a graphics controller **208** is coupled to bus bridge **202** through an AGP bus **210**. Finally, a plurality of PCI devices **212**A–**212**B are coupled to bus bridge **202** through a PCI bus **214**. A secondary bus bridge **216** may further be provided to accommodate an electrical interface to one or more EISA or ISA devices **218** through an EISA/ISA bus **220**. Processor **10** is coupled to bus bridge **202** through bus interface **46**.

Bus bridge **202** provides an interface between processor **10**, main memory **204**, graphics controller **208**, and devices attached to PCI bus **214**. When an operation is received from one of the devices connected to bus bridge **202**, bus bridge **202** identifies the target of the operation (e.g. a particular device or, in the case of PCI bus **214**, that the target is on PCI bus **214**). Bus bridge **202** routes the operation to the targeted device. Bus bridge **202** generally translates an operation from the protocol used by the source device or bus to the protocol used by the target device or bus.

In addition to providing an interface to an ISA/EISA bus for PCI bus **214**, secondary bus bridge **216** may further incorporate additional functionality, as desired. For example, in one embodiment, secondary bus bridge **216** includes a master PCI arbiter (not shown) for arbitrating ownership of PCI bus **214**. An input/output controller (not shown), either external from or integrated with secondary bus bridge **216**, may also be included within computer system **200** to provide operational support for a keyboard and mouse **222** and for various serial and parallel ports, as desired. An external cache unit (not shown) may further be coupled to bus interface **46** between processor **10** and bus bridge **202** in other embodiments. Alternatively, the external cache may be coupled to bus bridge **202** and cache control logic for the external cache may be integrated into bus bridge **202**.

Main memory **204** is a memory in which application programs are stored and from which processor **10** primarily executes. A suitable main memory **204** comprises DRAM (Dynamic Random Access Memory), and preferably a plurality of banks of SDRAM (Synchronous DRAM).

PCI devices **212**A–**212**B are illustrative of a variety of peripheral devices such as, for example, network interface cards, video accelerators, audio cards, hard or floppy disk drives or drive controllers, SCSI (Small Computer Systems Interface) adapters and telephony cards. Similarly, ISA device **218** is illustrative of various types of peripheral devices, such as a modem, a sound card, and a variety of data acquisition cards such as GPIB or field bus interface cards.

Graphics controller **208** is provided to control the rendering of text and images on a display **226**. Graphics controller **208** may embody a typical graphics accelerator generally known in the art to render three-dimensional data structures which can be effectively shifted into and from main memory **204**. Graphics controller **208** may therefore be a master of AGP bus **210** in that it can request and receive access to a target interface within bus bridge **202** to thereby obtain access to main memory **204**. A dedicated graphics bus accommodates rapid retrieval of data from main memory **204**. For certain operations, graphics controller **208** may further be configured to generate PCI protocol transactions on AGP bus **210**. The AGP interface of bus bridge **202** may thus include functionality to support both AGP protocol transactions as well as PCI protocol target and initiator transactions. Display **226** is any electronic display upon which an image or text can be presented. A suitable display **226** includes a cathode ray tube ("CRT"), a liquid crystal display ("LCD"), etc.

It is noted that, while the AGP, PCI, and ISA or EISA buses have been used as examples in the above description, any bus architectures may be substituted as desired. It is further noted that computer system **200** may be a multiprocessing computer system including additional processors (e.g. processor **10***a* shown as an optional component of computer system **200**). Processor **10***a* may be similar to processor **10**. More particularly, processor **10***a* may be an identical copy of processor **10**. Processor **10***a* may share bus interface **46** with processor **10** (as shown in FIG. **15**) or may be connected to bus bridge **202** via an independent bus.

Numerous variations and modifications will become apparent to those skilled in the art once the above disclosure is fully appreciated. It is intended that the following claims be interpreted to embrace all such variations and modifications.

What is claimed is:

1. A processor comprising:

a predecode unit configured to predecode a plurality of instruction bytes received by said processor, wherein said predecode unit, upon predecoding a relative control transfer instruction comprising a displacement, is configured to add an address to said displacement to generate a target address corresponding to said relative control transfer instruction, and wherein said predecode unit is configured to replace said displacement within said relative control transfer instruction with an encoded value indicative of said target address, and wherein a control transfer instruction, when executed, specifies an address from which a subsequent instruction to be executed is fetched, and wherein said predecode unit is configured to generate a plurality of control transfer indications, and wherein each one of said plurality of control transfer indications corresponds to a different one of said plurality of instruction bytes, and wherein said plurality of control transfer indications identify control transfer instructions including said relative control transfer instruction; and

an instruction cache coupled to said predecode unit, wherein said instruction cache is configured to store said plurality of instruction bytes including said relative control transfer instruction with said encoded value in place of said displacement, and wherein said instruction cache is configured to store said plurality of control transfer indications.

2. The processor as recited in claim **1** wherein said displacement includes fewer bits than said target address.

3. The processor as recited in claim **2** wherein said encoded value includes a first field and a second field.

4. The processor as recited in claim **4** wherein said first field comprises an offset within a target cache line of a byte identified by said target address.

5. A method for generating a target address for a relative control transfer instruction, the method comprising:

predecoding a plurality of instruction bytes including said relative transfer instruction to detect a presence of said relative control transfer instruction, wherein a control transfer instruction, when executed, specifies an address from which a subsequent instruction to be executed is fetched, and wherein said predecoding comprises generating a plurality of control transfer indications, wherein each of said plurality of control transfer indications corresponds to a different one of said plurality of instruction bytes, and wherein said plurality of control transfer indications identify control transfer instructions starting at said different ones of said plurality of instruction bytes;

adding an address to a displacement included in said relative control transfer instruction, thereby generating said target address;

replacing said displacement within said relative control transfer instruction with an encoding indicative of said target address; and

storing said plurality of instruction bytes including said relative control transfer instruction, with said displacement replaced by said encoding, and said plurality of control transfer indications in an instruction cache.

6. A predecode unit comprising:

a decoder configured to decode a plurality of instruction bytes and to identify a relative control transfer instruction therein, wherein a control transfer instruction, when executed, specifies an address from which a subsequent instruction to be executed is fetched, and wherein said decoder is configured to generate a plurality of control transfer indications, and wherein each one of said plurality of control transfer indications corresponds to a different one of said plurality of instruction bytes, and wherein said plurality of control transfer indications identify control transfer instructions including said relative control transfer instruction; and

a target generator configured to add a displacement selected from said relative control transfer instruction to an address, thereby generating a target address corresponding to said relative control transfer instruction, and further configured to generate an encoding of said target address with which said predecode unit replaces said displacement within said relative control transfer instruction.

7. The predecode unit as recited in claim **6** wherein said target generator includes a sign extend block configured to sign extend said displacement to a number of bits in said address.

8. The predecode unit as recited in claim **7** wherein said target generator further includes:

an adder coupled to said sign extend block and to receive said address, wherein said adder is configured to add said sign extended displacement and said address to generate said target address; and

a displacement encoder coupled to said adder, wherein said displacement encoder is configured to encode said target address.

9. The predecode unit as recited in claim **8** wherein said displacement encoder is configured to encode said target address as: (i) an offset within a target cache line of a byte identified by said target address; and (ii) a number of cache lines above or below a cache line storing said relative control transfer instruction at which said target cache line is stored.

10. A computer system comprising:

a processor configured to predecode a plurality of instruction bytes received by said processor, wherein said processor, upon predecoding a relative control transfer instruction comprising a displacement, is configured to add an address to said displacement to generate a target address corresponding to said relative control transfer instruction, and wherein said processor is configured to replace said displacement within said relative control transfer instruction with an encoded value indicative of said target address, and wherein a control transfer instruction, when executed, specifies an address from which a subsequent instruction to be executed is fetched, and wherein said processor, during predecoding, is configured to generate a plurality of control transfer indications, and wherein each one of said plurality of control transfer indications corresponds to a different one of said plurality of instruction bytes, and wherein said plurality of control transfer indications identify control transfer instructions including said relative control transfer instruction;

a memory coupled to said processor, wherein said memory is configured to store said plurality of instruction bytes and to provide said instruction bytes to said processor; and

a peripheral device configured to transfer data between said computer system and another computer system.

11. The computer system as recited in claim **10** wherein said peripheral device is a modem.

12. The computer system as recited in claim **10** further comprising an audio peripheral device.

13. The computer system as recited in claim **12** wherein said audio I/O device comprises a sound card.

14. The computer system as recited in claim **10** further comprising a second processor configured to predecode a plurality of instruction bytes received by said second processor, wherein said second processor, upon predecoding a relative control transfer instruction comprising a displacement, is configured to add an address to said displacement to generate a target address corresponding to said relative control transfer instruction, and wherein said second processor is configured to replace said displacement within said relative control transfer instruction with an encoded value indicative of said target address.

* * * * *

US006496187B1

(12) **United States Patent**
Deering et al.

(10) **Patent No.: US 6,496,187 B1**
(45) **Date of Patent: Dec. 17, 2002**

(54) GRAPHICS SYSTEM CONFIGURED TO PERFORM PARALLEL SAMPLE TO PIXEL CALCULATION

(75) Inventors: **Michael F. Deering**, Los Altos, CA (US); **Nathaniel David Naegle**, Pleasanton, CA (US); **Scott R. Nelson**, Pleasanton, CA (US)

(73) Assignee: **Sun Microsystems, Inc.**, Palo Alto, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/472,940**

(22) Filed: **Dec. 27, 1999**

Related U.S. Application Data

(63) Continuation-in-part of application No. 09/251,844, filed on Feb. 17, 1999.
(60) Provisional application No. 60/074,836, filed on Feb. 17, 1998.

(51) **Int. Cl.**^{7} **G06T 15/00**
(52) **U.S. Cl.** **345/419**; 345/611
(58) **Field of Search** 345/419, 420, 345/426, 427, 428, 581, 589, 611, 629, 612, 613, 614, 615, 473

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,117,289	A		5/1992	Farley et al.	
5,287,438	A	*	2/1994	Kelleher	345/613
5,481,669	A	*	1/1996	Poulton et al.	345/505
5,619,438	A		4/1997	Farley et al.	
5,742,277	A	*	4/1998	Gossett et al.	345/611
5,745,125	A	*	4/1998	Deering et al.	345/426
5,999,187	A	*	12/1999	Dehmlow et al.	345/420
6,072,498	A	*	6/2000	Brittain et al.	345/428
6,204,859	B1	*	3/2001	Jouppi et al.	345/422
6,313,838	B1	*	11/2001	Deering	345/420

FOREIGN PATENT DOCUMENTS

EP	0 463 700	1/1992
GB	2 278 524	11/1994
WO	91/14995	10/1991

OTHER PUBLICATIONS

International Search Report for Application No. PCT/US 00/04148, mailed May 29, 2000.
Hadfield et al., "Achieving Real–Time Visual Simulation Using PC Graphics Technology," presented at I/ITSEC Nov. 1999.
Bjernfalk, "Introducing REALimage™ 4000 and HYPERpixel™ Architecture," © 1999 Evans & Sutherland Computer Corporation, pp. 1–9.
Bjernfalk, "The Memory System Makes the Difference," © 1999 Evans & Sutherland Computer Corporation, pp. 1–11.

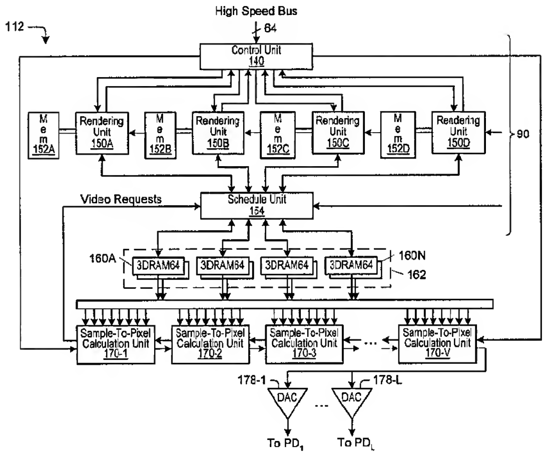
* cited by examiner

Primary Examiner—Mark Zimmerman
Assistant Examiner—Enrique L Santiago
(74) *Attorney, Agent, or Firm*—Jeffrey C. Hood

(57) **ABSTRACT**

A graphics system that is configured to utilize a sample buffer and a plurality of parallel sample-to-pixel calculation units, wherein the sample-pixel calculation units are configured to access different portions of the sample buffer in parallel. The graphics system may include a graphics processor, a sample buffer, and a plurality of sample-to-pixel calculation units. The graphics processor is configured to receive a set of three-dimensional graphics data and render a plurality of samples based on the graphics data. The sample buffer is configured to store the plurality of samples for the sample-to-pixel calculation units, which are configured to receive and filter samples from the sample buffer to create output pixels. Each of the sample-to-pixel calculation units are configured to generate pixels corresponding to a different region of the image. The region may be a vertical or horizontal stripe of the image, or a rectangular portion of the image. Each region may overlap the other regions of the image to prevent visual aberrations.

52 Claims, 25 Drawing Sheets



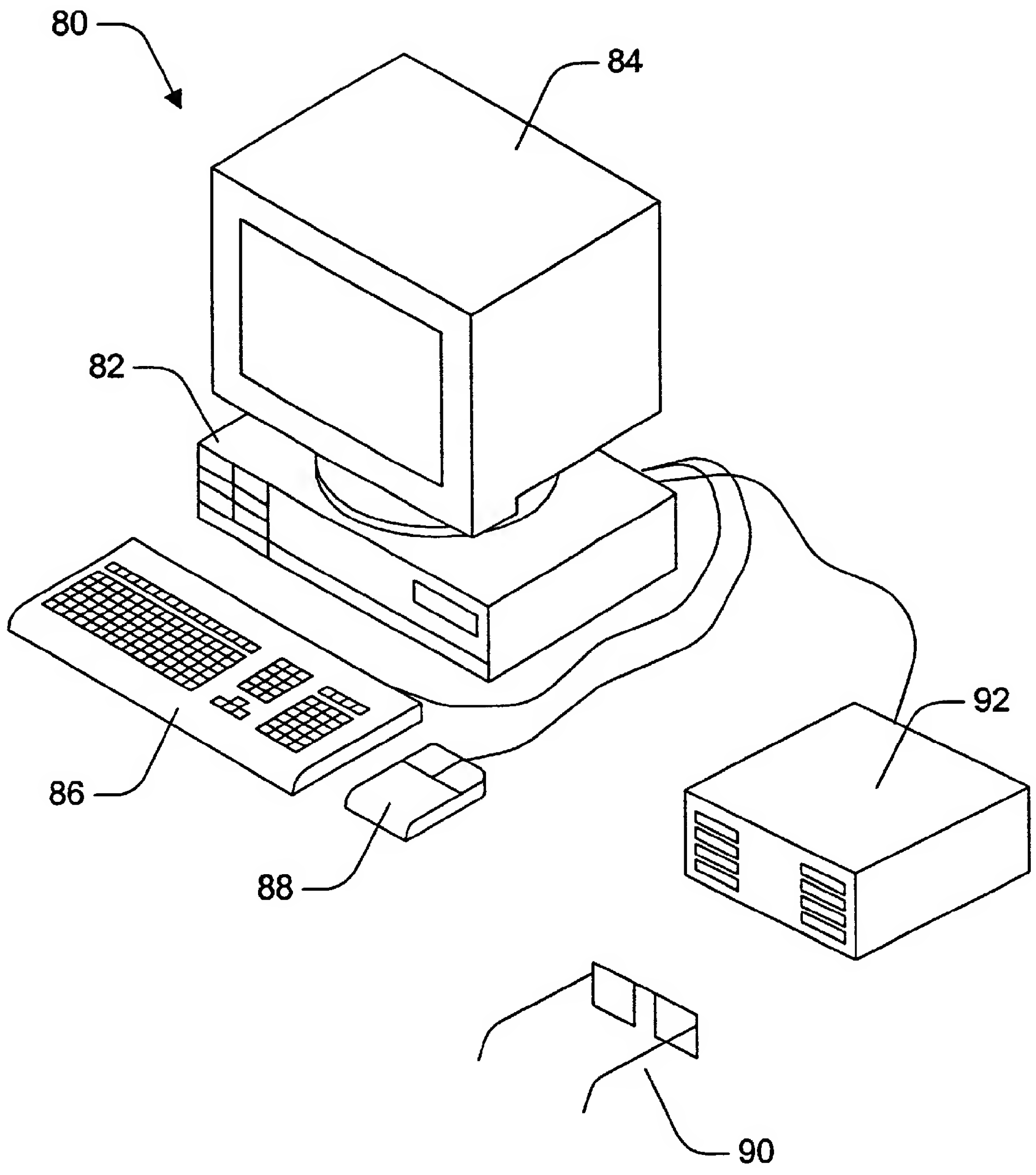


FIG. 1A

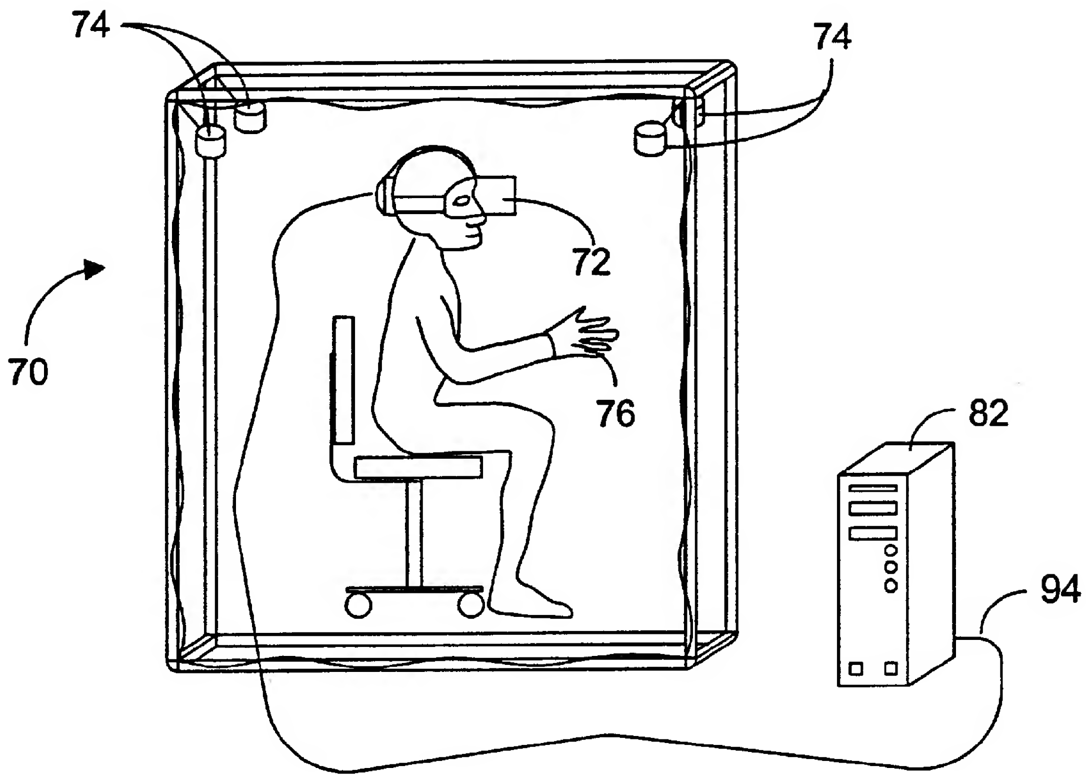


FIG. 1B

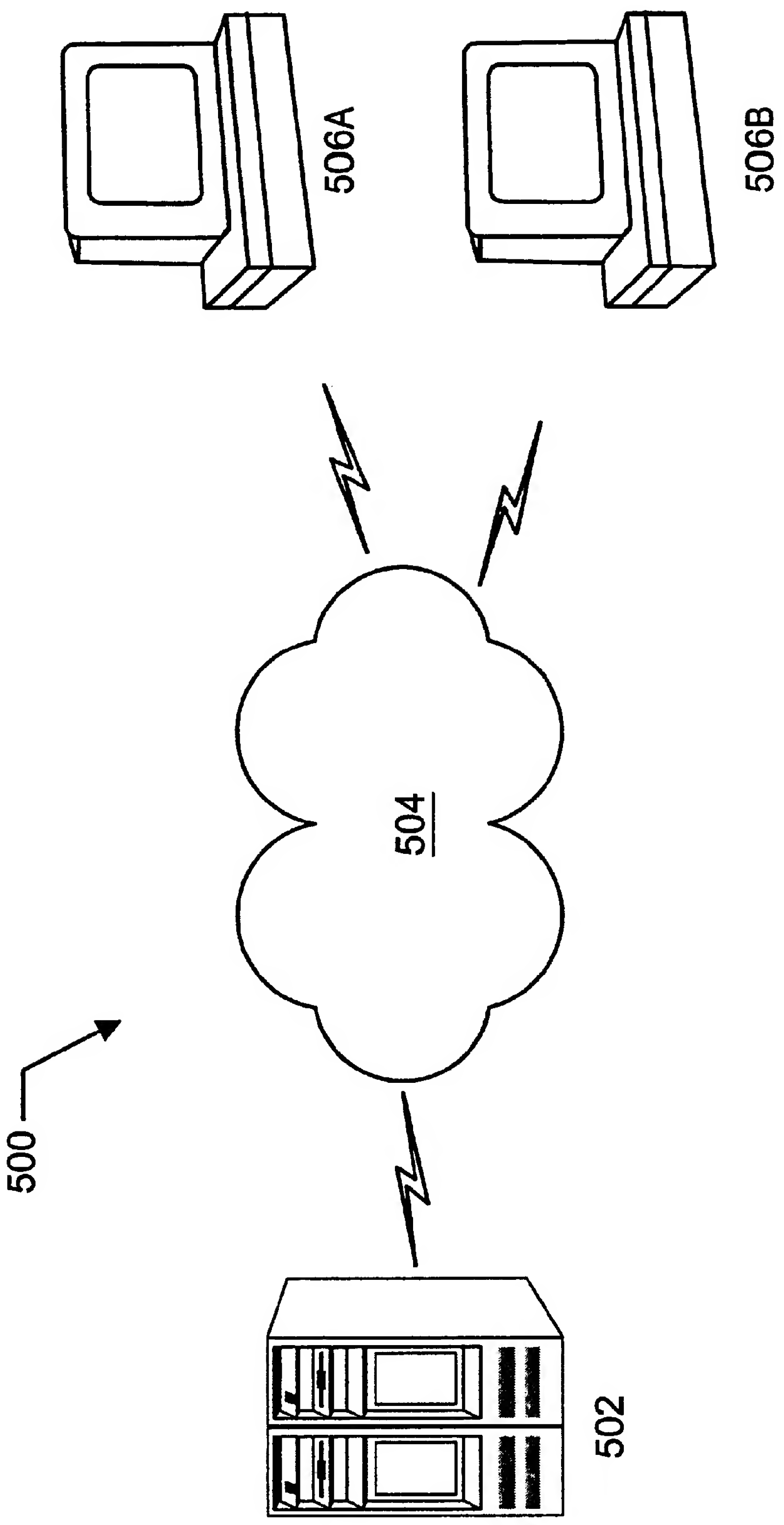


FIG. 2

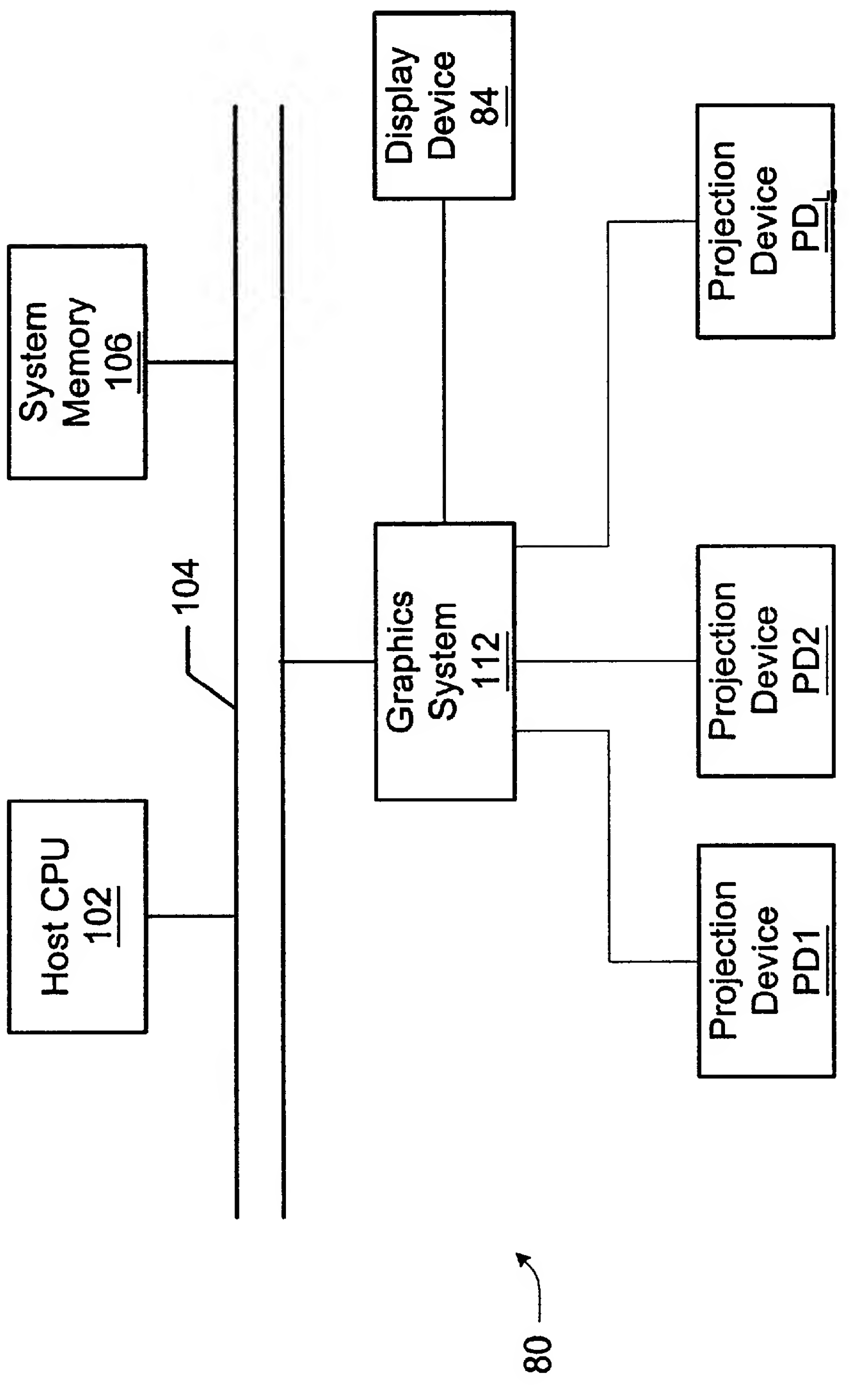

Host CPU 102
System Memory 106
104
Graphics System 112
Display Device 84
Projection Device PD1
Projection Device PD2
Projection Device PD_L
80

FIG. 3A

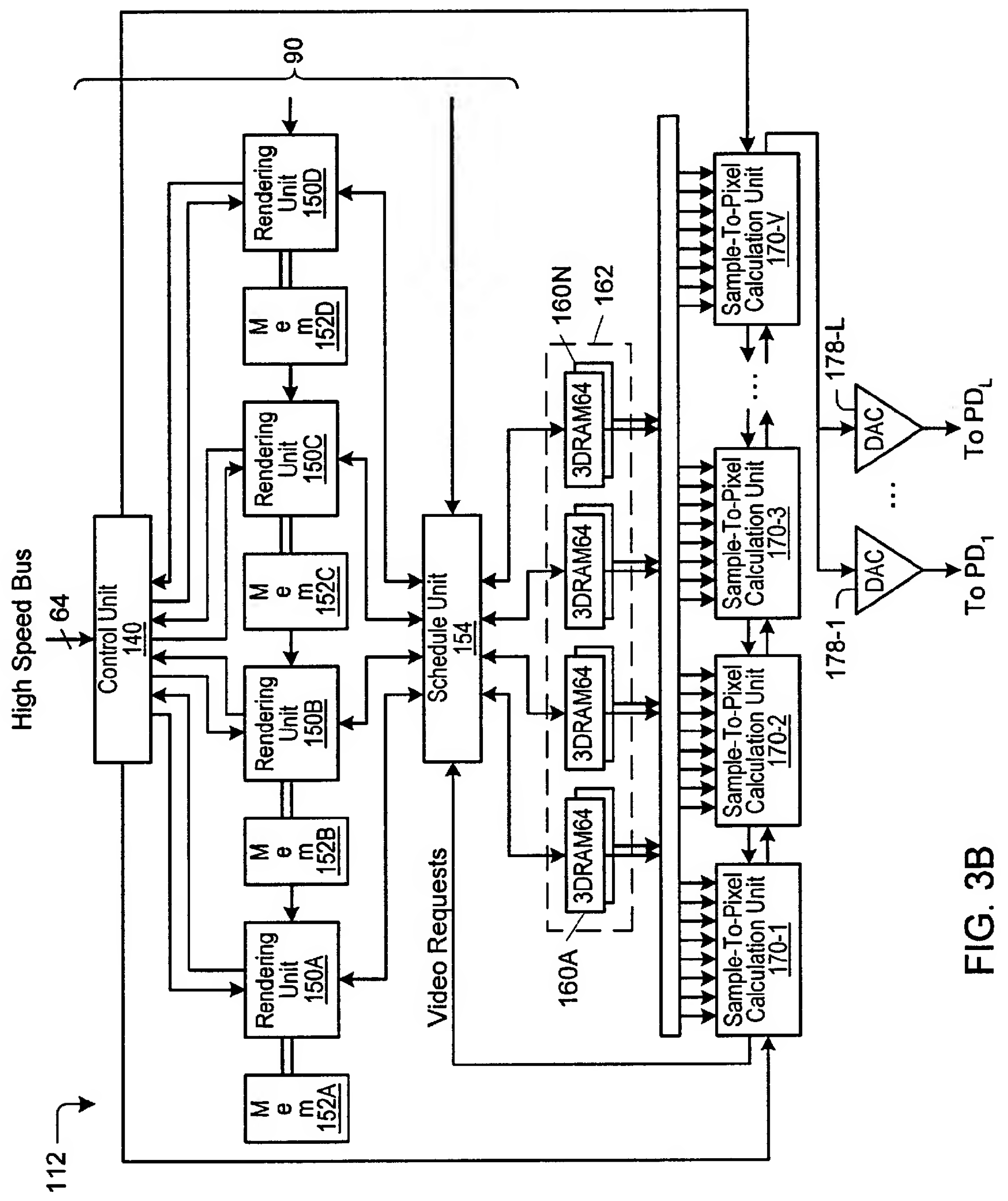

112
High Speed Bus
64
Control Unit 140
90
Rendering Unit 150A
Rendering Unit 150B
Rendering Unit 150C
Rendering Unit 150D
Mem 152A
Mem 152B
Mem 152C
Mem 152D
Schedule Unit 154
Video Requests
160A
160N
162
3DRAM64
3DRAM64
3DRAM64
3DRAM64
Sample-To-Pixel Calculation Unit 170-1
Sample-To-Pixel Calculation Unit 170-2
Sample-To-Pixel Calculation Unit 170-3
Sample-To-Pixel Calculation Unit 170-V
178-1
178-L
DAC
DAC
To PD_1
To PD_L

FIG. 3B

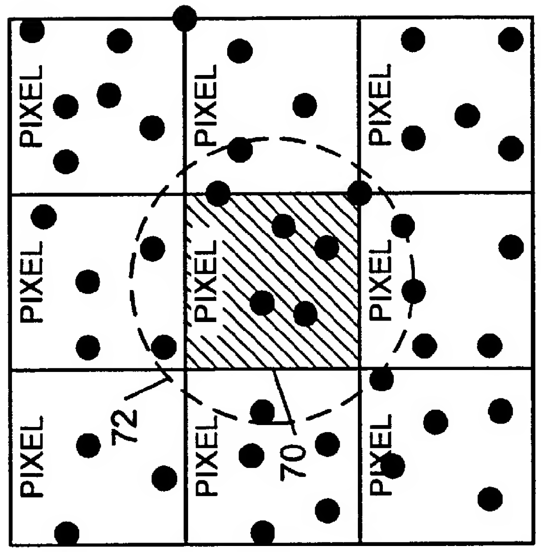


FIG. 5B

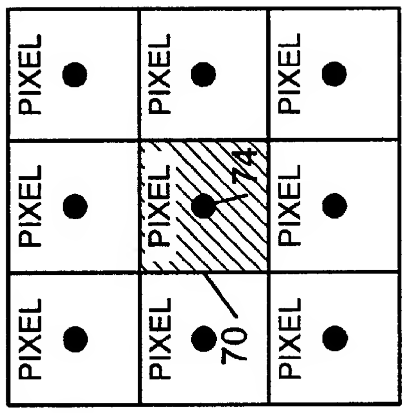


FIG. 4

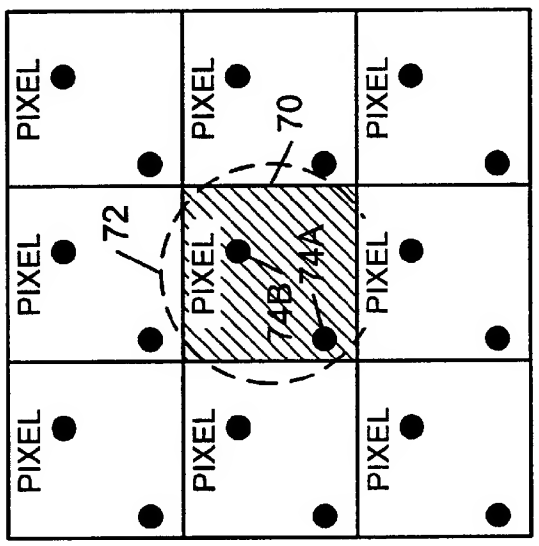


FIG. 5A

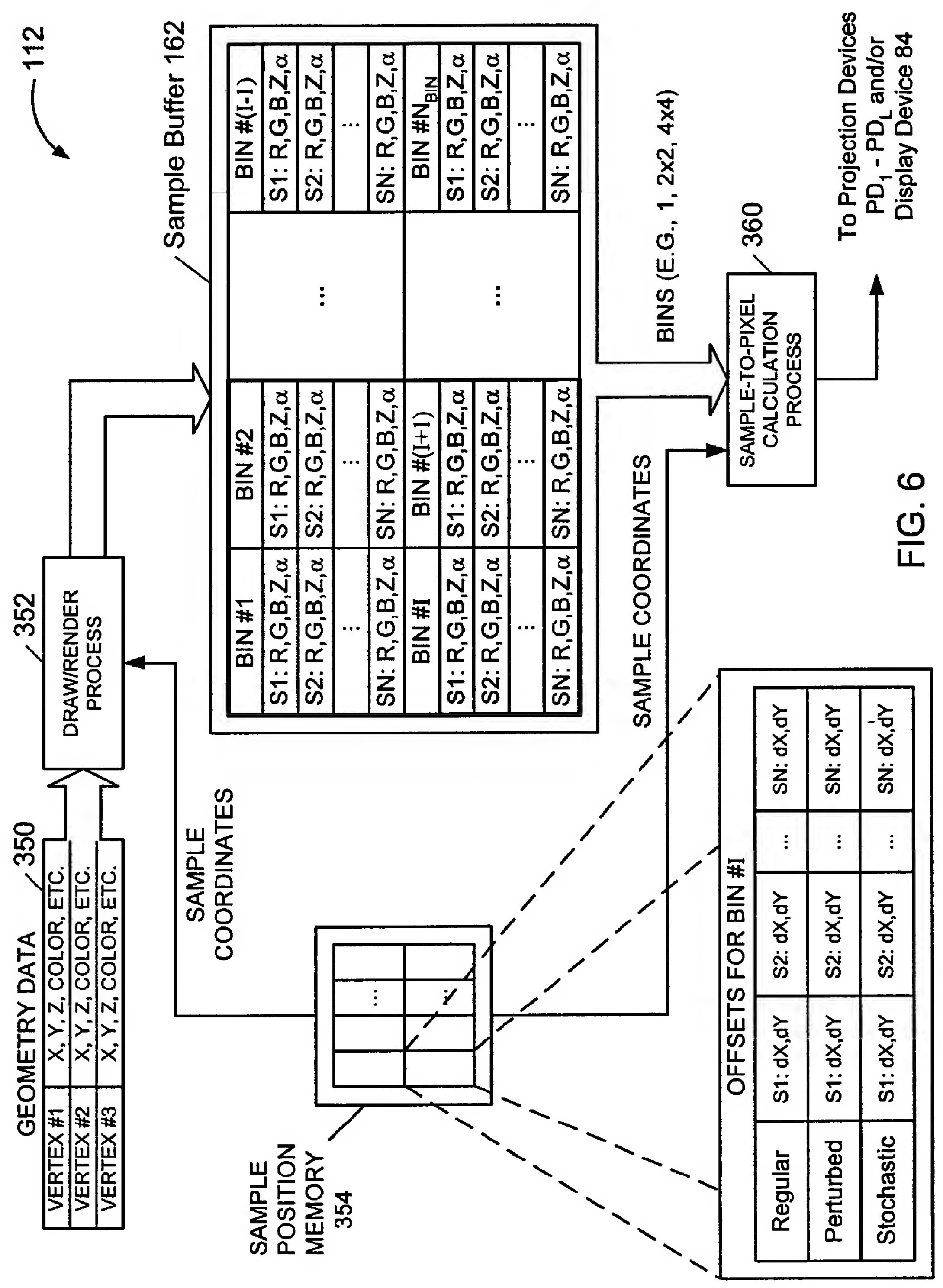

112
GEOMETRY DATA
350
VERTEX #1 X, Y, Z, COLOR, ETC.
VERTEX #2 X, Y, Z, COLOR, ETC.
VERTEX #3 X, Y, Z, COLOR, ETC.
352
DRAW/RENDER PROCESS
SAMPLE COORDINATES
Sample Buffer 162
BIN #1
S1: R,G,B,Z,α
S2: R,G,B,Z,α
...
SN: R,G,B,Z,α
BIN #2
BIN #(I-1)
BIN #I
BIN #(I+1)
BIN #N_{BIN}
SAMPLE POSITION MEMORY 354
SAMPLE COORDINATES
BINS (E.G., 1, 2x2, 4x4)
360
SAMPLE-TO-PIXEL CALCULATION PROCESS
To Projection Devices PD_1 - PD_L and/or Display Device 84
OFFSETS FOR BIN #I
Regular
Perturbed
Stochastic
S1: dX,dY
S2: dX,dY
...
SN: dX,dY

FIG. 6

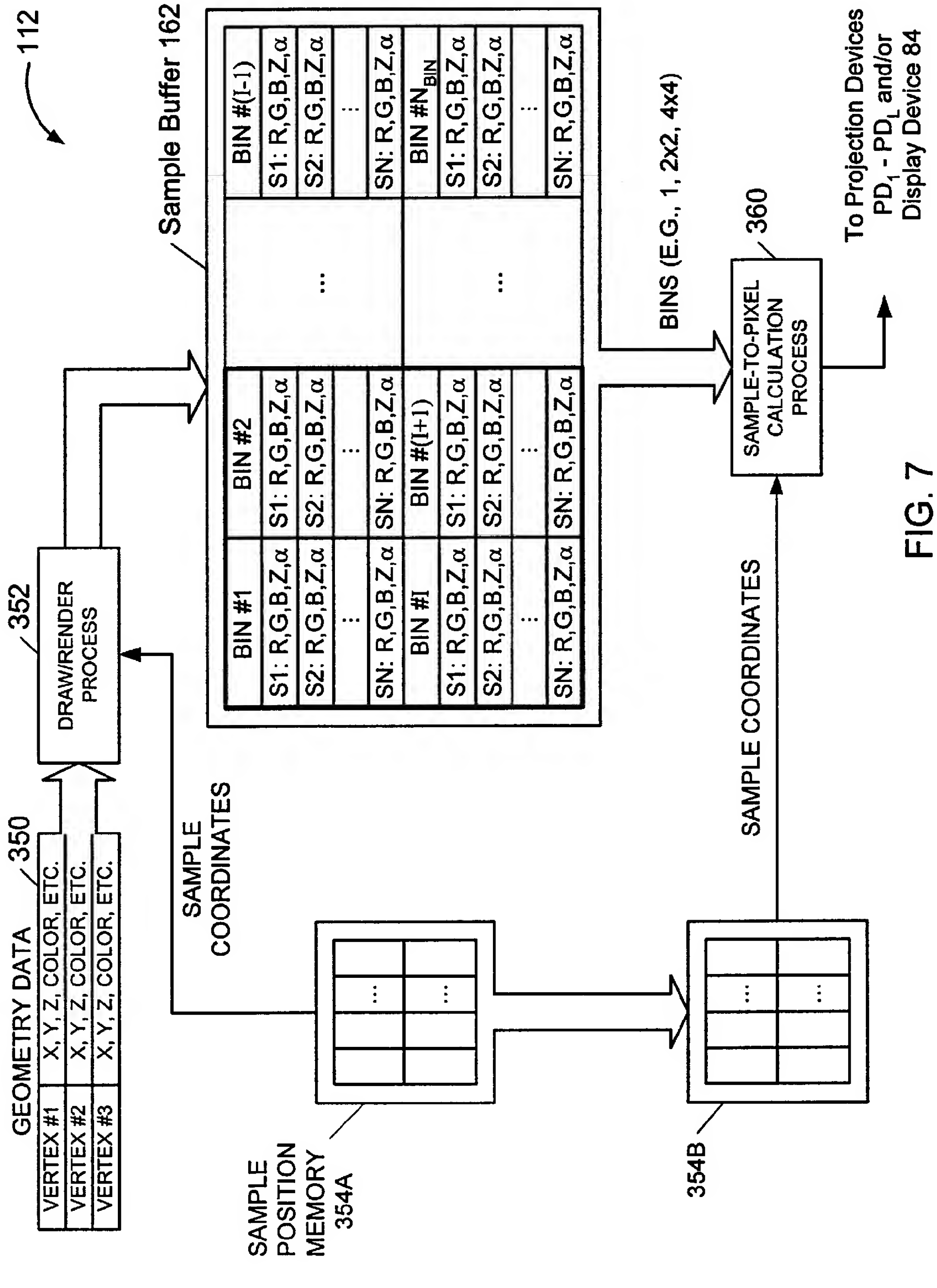

112
GEOMETRY DATA
350
VERTEX #1 X, Y, Z, COLOR, ETC.
VERTEX #2 X, Y, Z, COLOR, ETC.
VERTEX #3 X, Y, Z, COLOR, ETC.
352
DRAW/RENDER PROCESS
SAMPLE COORDINATES
Sample Buffer 162
BIN #1
S1: R,G,B,Z,α
S2: R,G,B,Z,α
SN: R,G,B,Z,α
BIN #2
BIN #(I-1)
BIN #I
BIN #(I+1)
BIN #N$_{BIN}$
BINS (E.G., 1, 2x2, 4x4)
360
SAMPLE-TO-PIXEL CALCULATION PROCESS
To Projection Devices PD$_1$ - PD$_L$ and/or Display Device 84
SAMPLE COORDINATES
SAMPLE POSITION MEMORY 354A
354B

FIG. 7

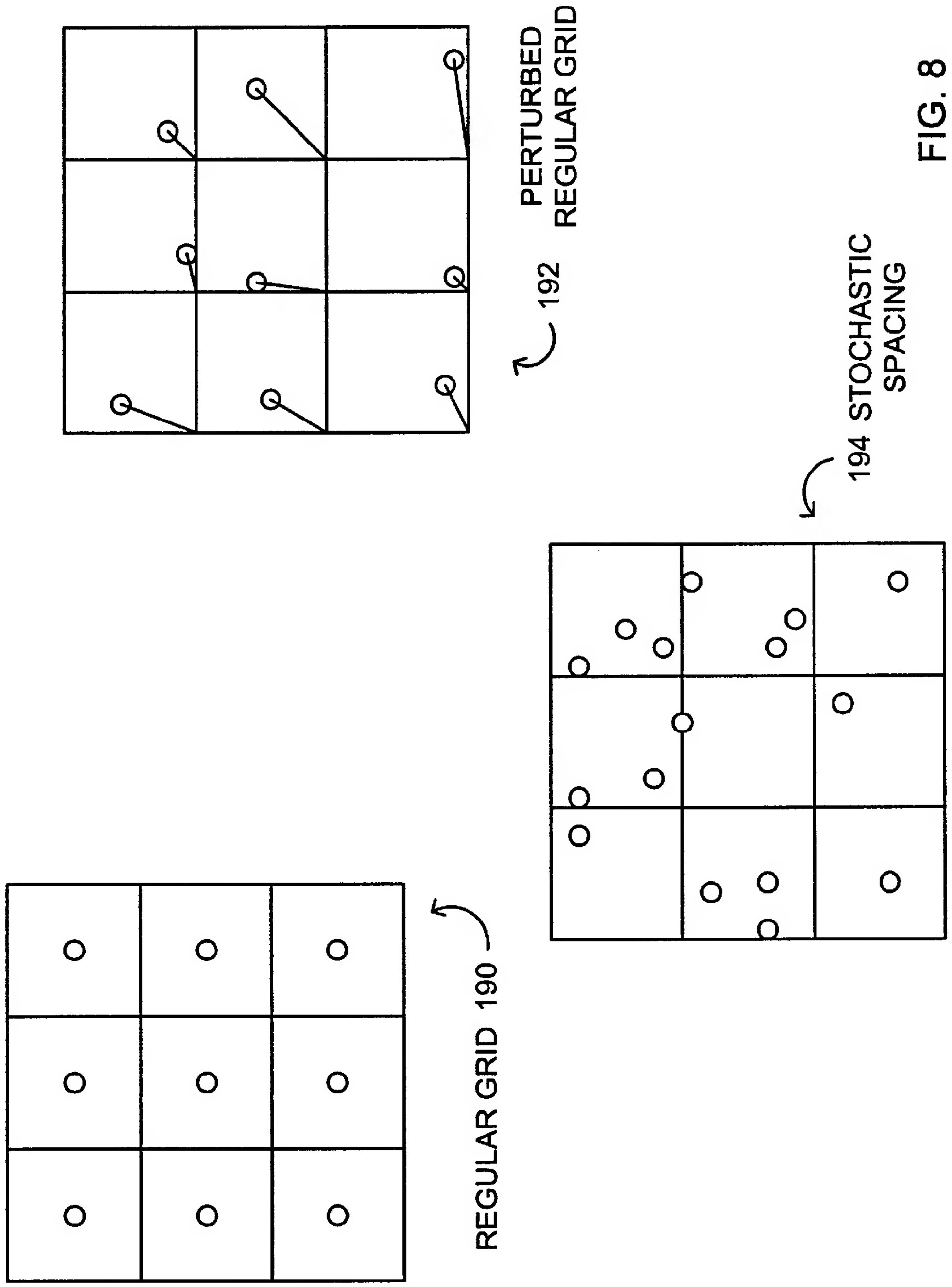

REGULAR GRID 190
192
PERTURBED REGULAR GRID
194 STOCHASTIC SPACING

FIG. 8

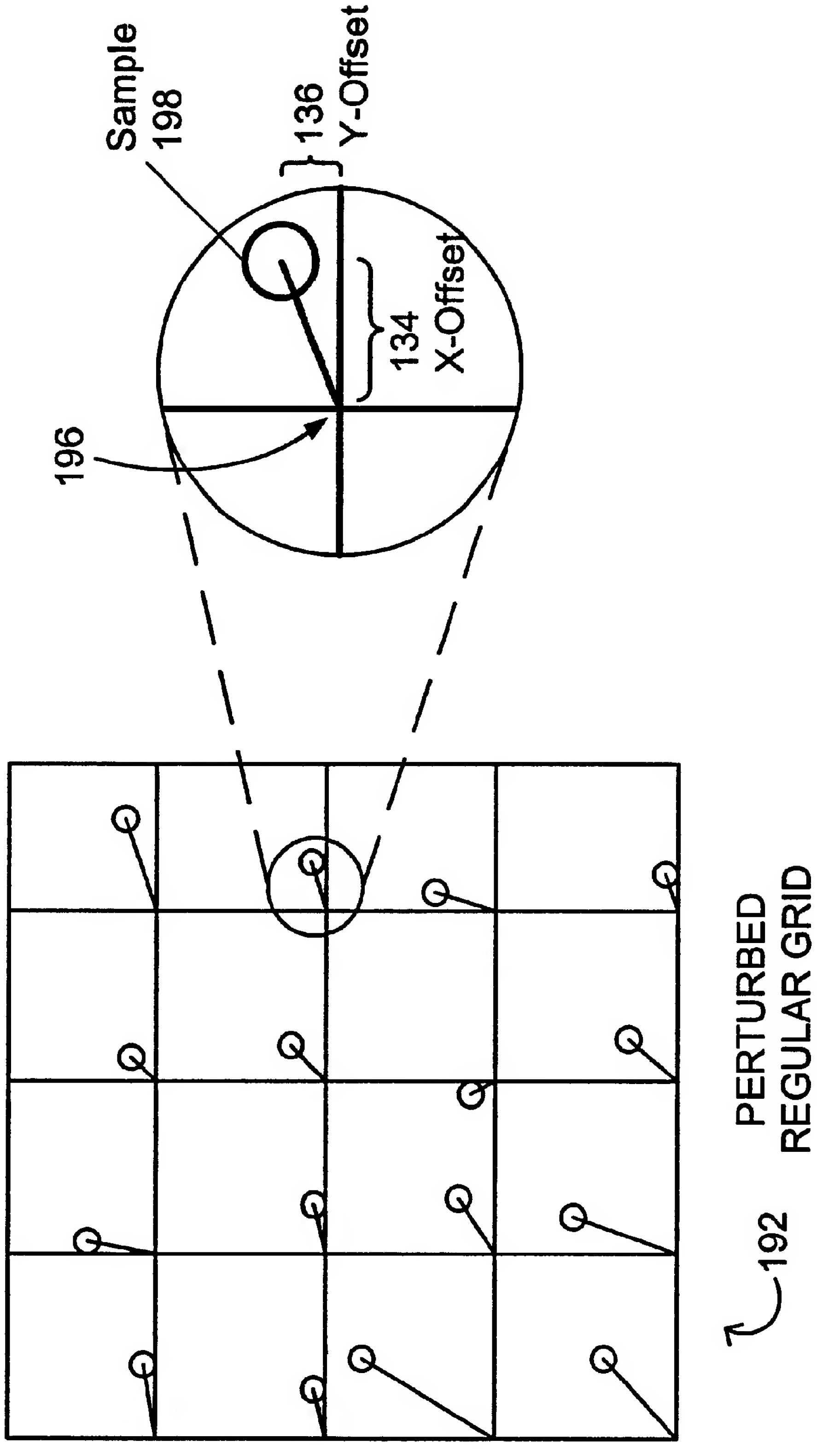

Sample
198
136
Y-Offset
134
X-Offset
196
PERTURBED
REGULAR GRID
192

FIG. 9

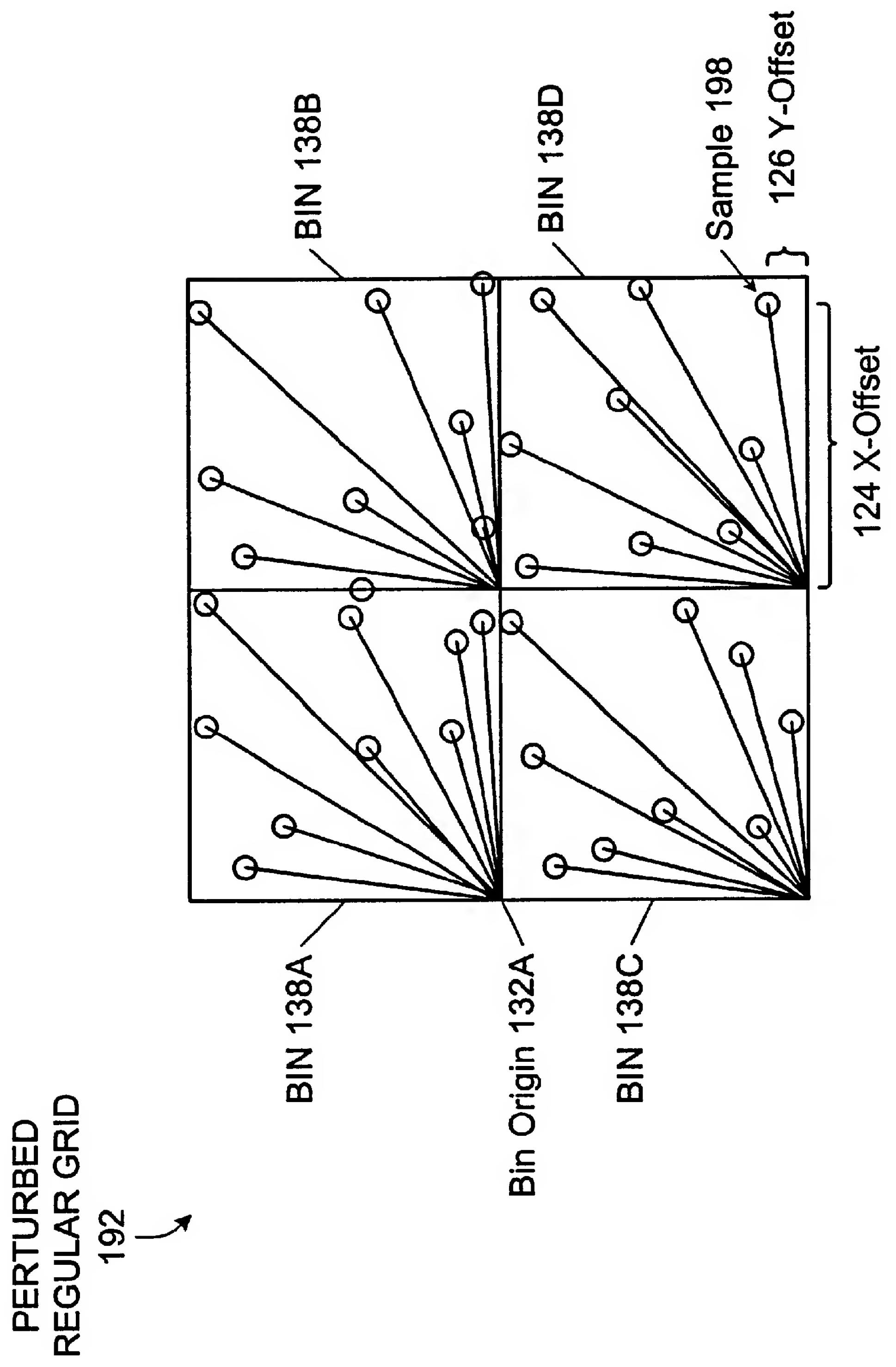

PERTURBED REGULAR GRID 192
BIN 138A
Bin Origin 132A
BIN 138C
BIN 138B
BIN 138D
Sample 198
126 Y-Offset
124 X-Offset

FIG. 10

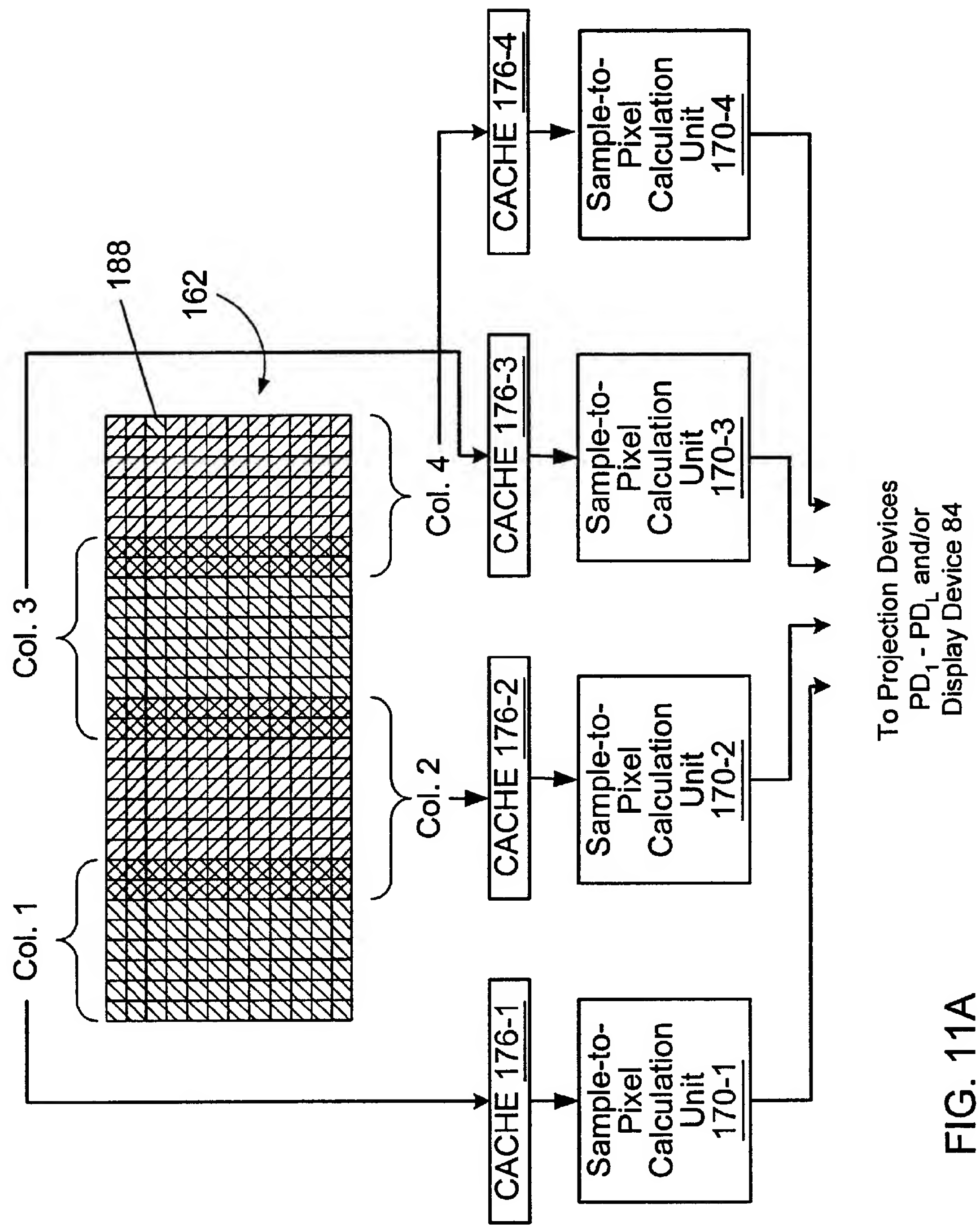

188
162
Col. 1
Col. 2
Col. 3
Col. 4
CACHE 176-1
CACHE 176-2
CACHE 176-3
CACHE 176-4
Sample-to-Pixel Calculation Unit 170-1
Sample-to-Pixel Calculation Unit 170-2
Sample-to-Pixel Calculation Unit 170-3
Sample-to-Pixel Calculation Unit 170-4
To Projection Devices PD_1 - PD_L and/or Display Device 84

FIG. 11A

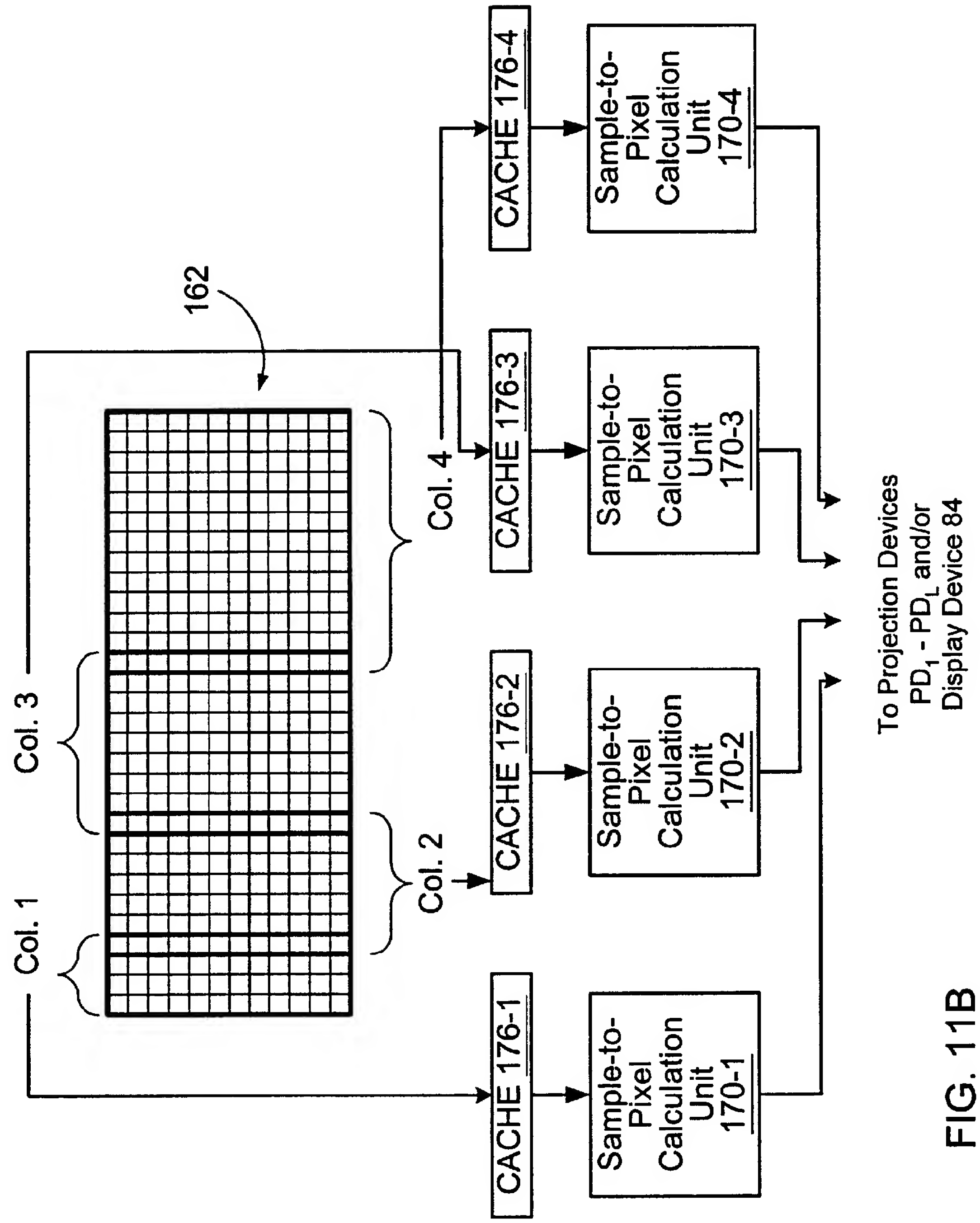

162
Col. 1
Col. 2
Col. 3
Col. 4
CACHE 176-1
CACHE 176-2
CACHE 176-3
CACHE 176-4
Sample-to-Pixel Calculation Unit 170-1
Sample-to-Pixel Calculation Unit 170-2
Sample-to-Pixel Calculation Unit 170-3
Sample-to-Pixel Calculation Unit 170-4
To Projection Devices PD_1 - PD_L and/or Display Device 84

FIG. 11B

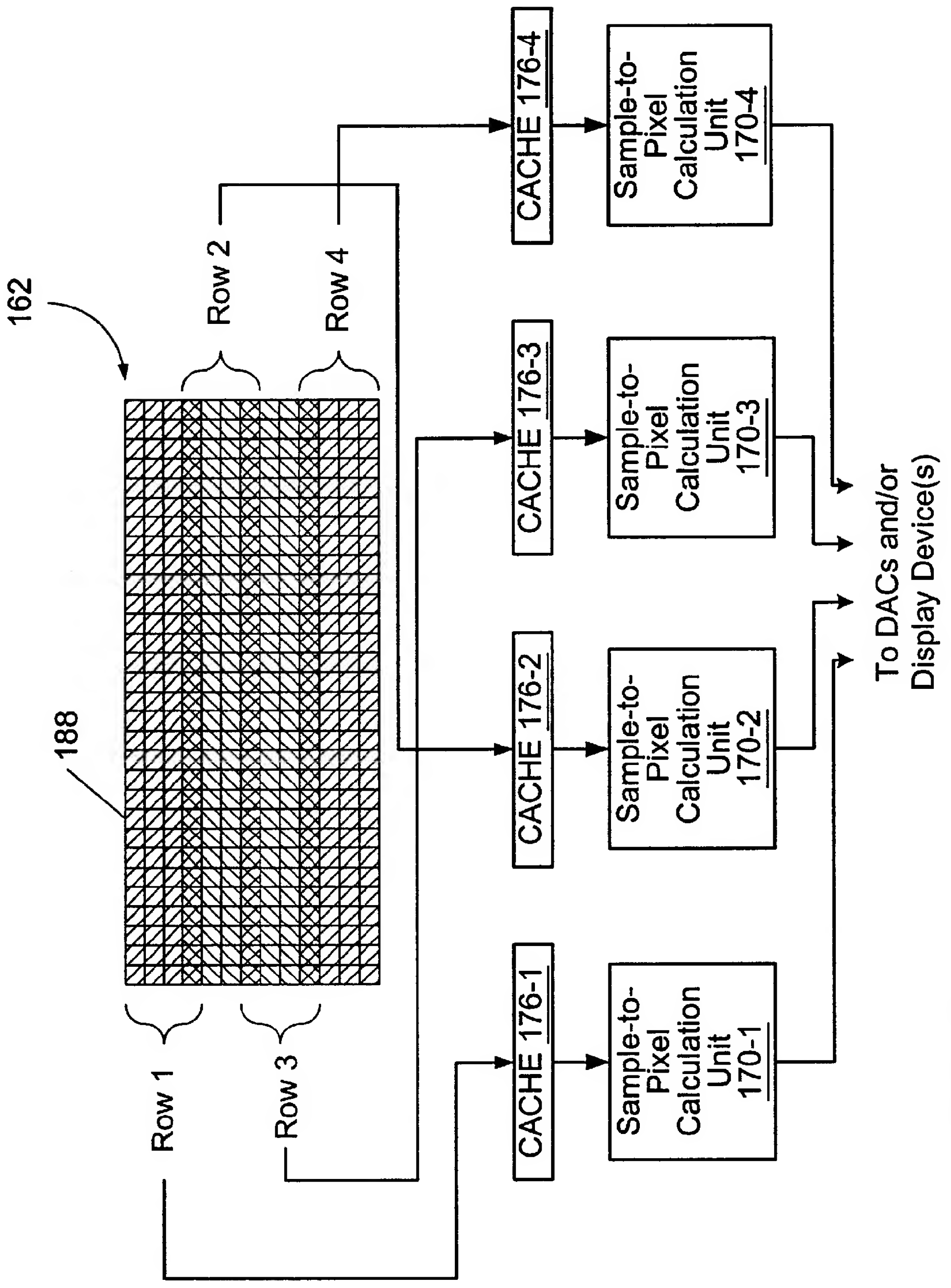

162
188
Row 1
Row 2
Row 3
Row 4
CACHE 176-1
CACHE 176-2
CACHE 176-3
CACHE 176-4
Sample-to-Pixel Calculation Unit 170-1
Sample-to-Pixel Calculation Unit 170-2
Sample-to-Pixel Calculation Unit 170-3
Sample-to-Pixel Calculation Unit 170-4
To DACs and/or Display Device(s)

FIG. 12

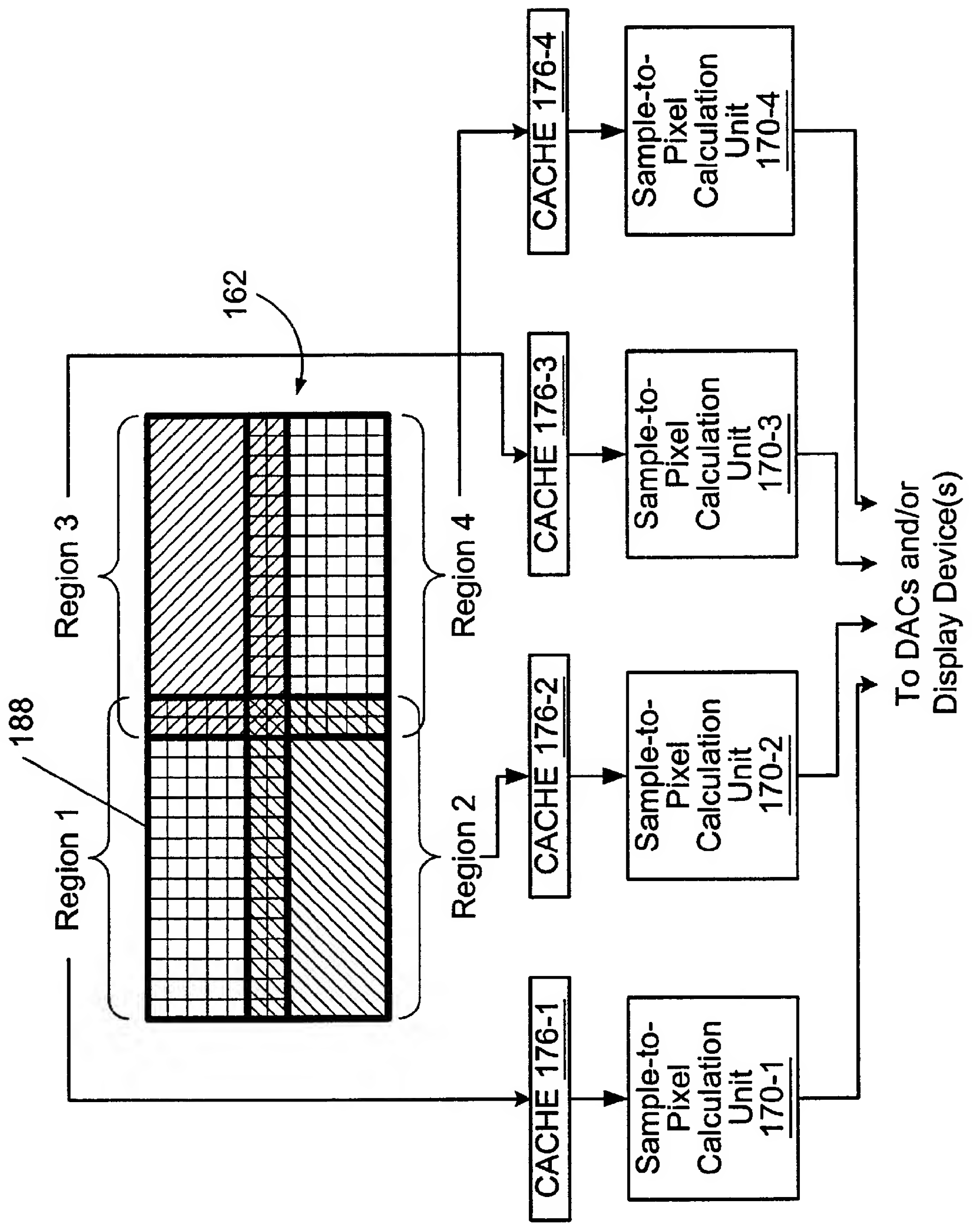

188
162
Region 1
Region 3
Region 2
Region 4
CACHE 176-1
CACHE 176-2
CACHE 176-3
CACHE 176-4
Sample-to-Pixel Calculation Unit 170-1
Sample-to-Pixel Calculation Unit 170-2
Sample-to-Pixel Calculation Unit 170-3
Sample-to-Pixel Calculation Unit 170-4
To DACs and/or Display Device(s)

FIG. 13

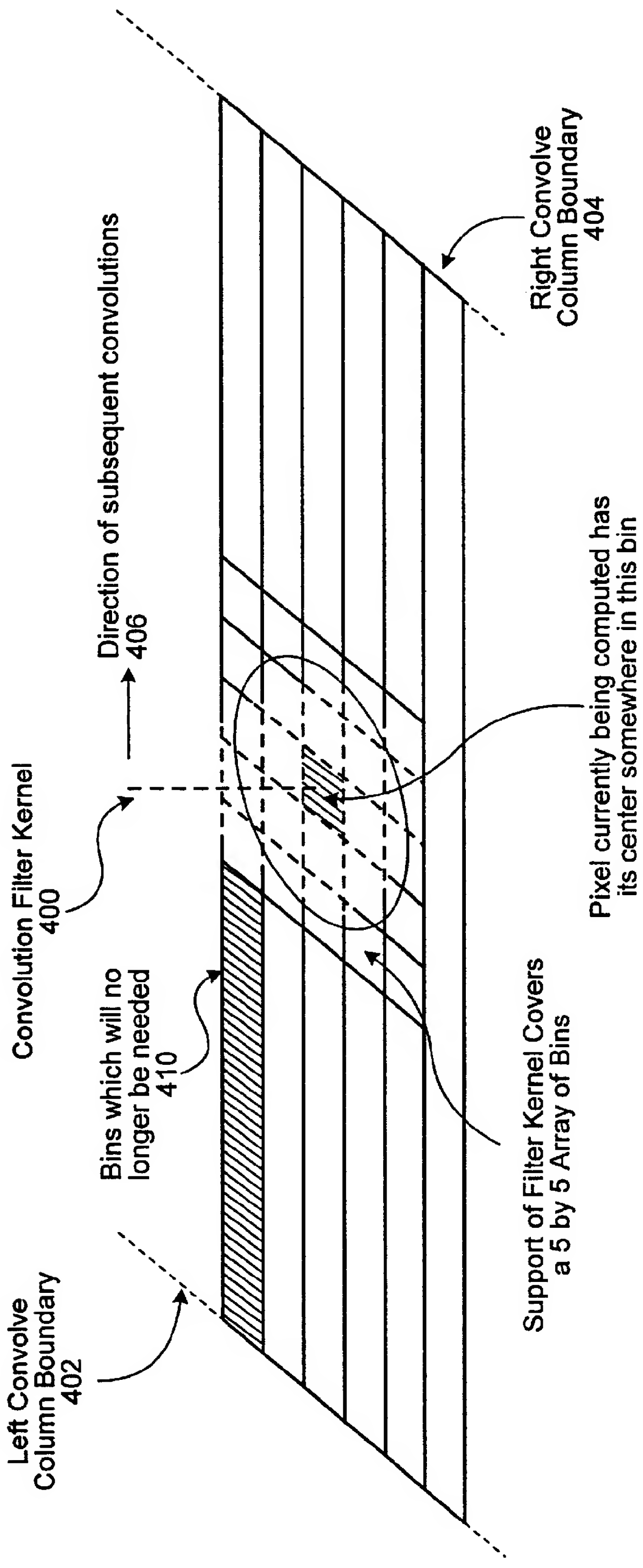

Direction of subsequent convolutions
Right Convolve Column Boundary 404
Direction 406
Convolution Filter Kernel 400
Pixel currently being computed has its center somewhere in this bin
Bins which will no longer be needed 410
Support of Filter Kernel Covers a 5 by 5 Array of Bins
Left Convolve Column Boundary 402

FIG. 14

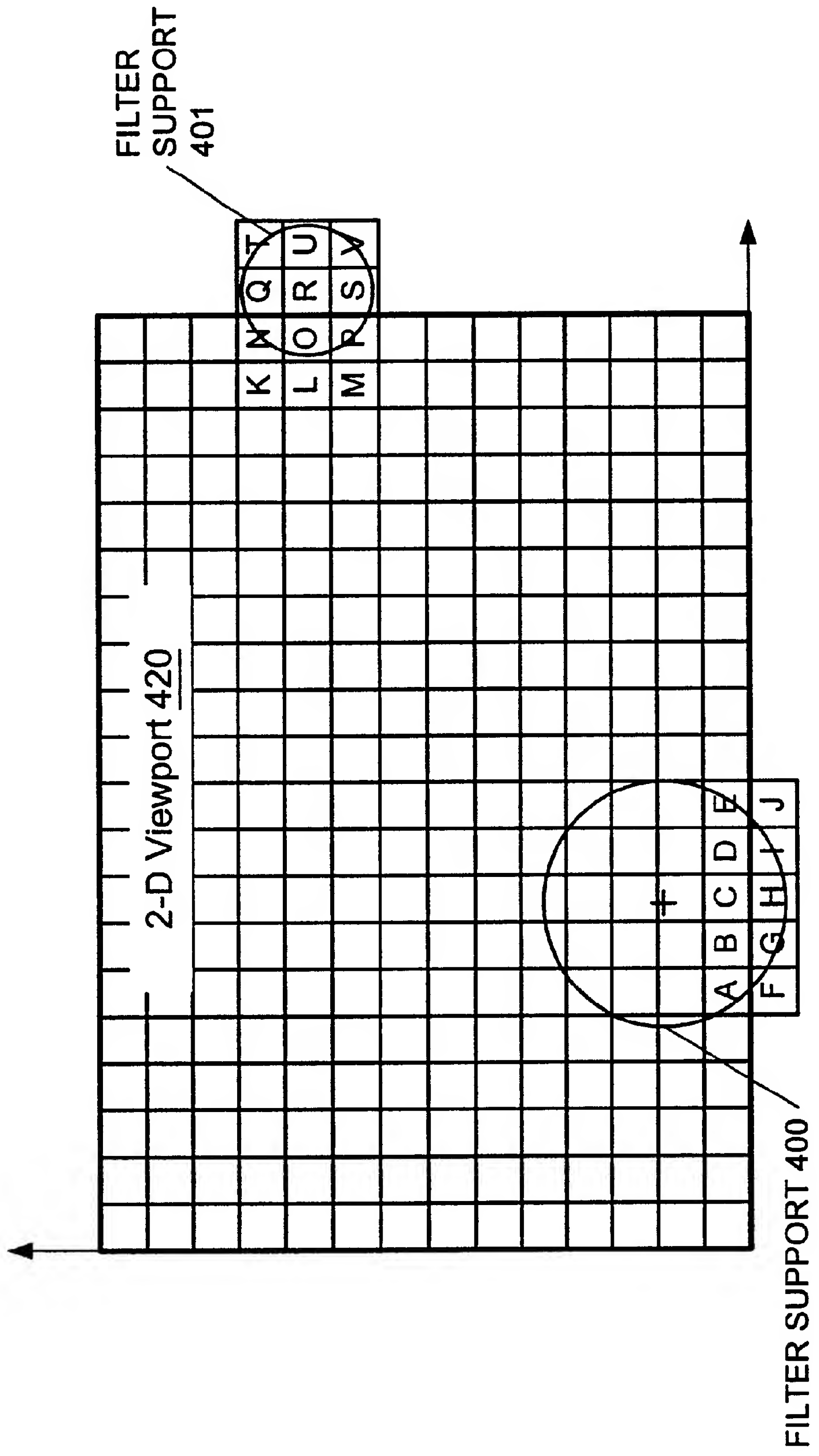

FILTER SUPPORT 401
K
L
M
N
O
P
Q
R
S
T
U
V
2-D Viewport 420
A
B
C
D
E
F
G
H
I
J
FILTER SUPPORT 400

FIG. 15

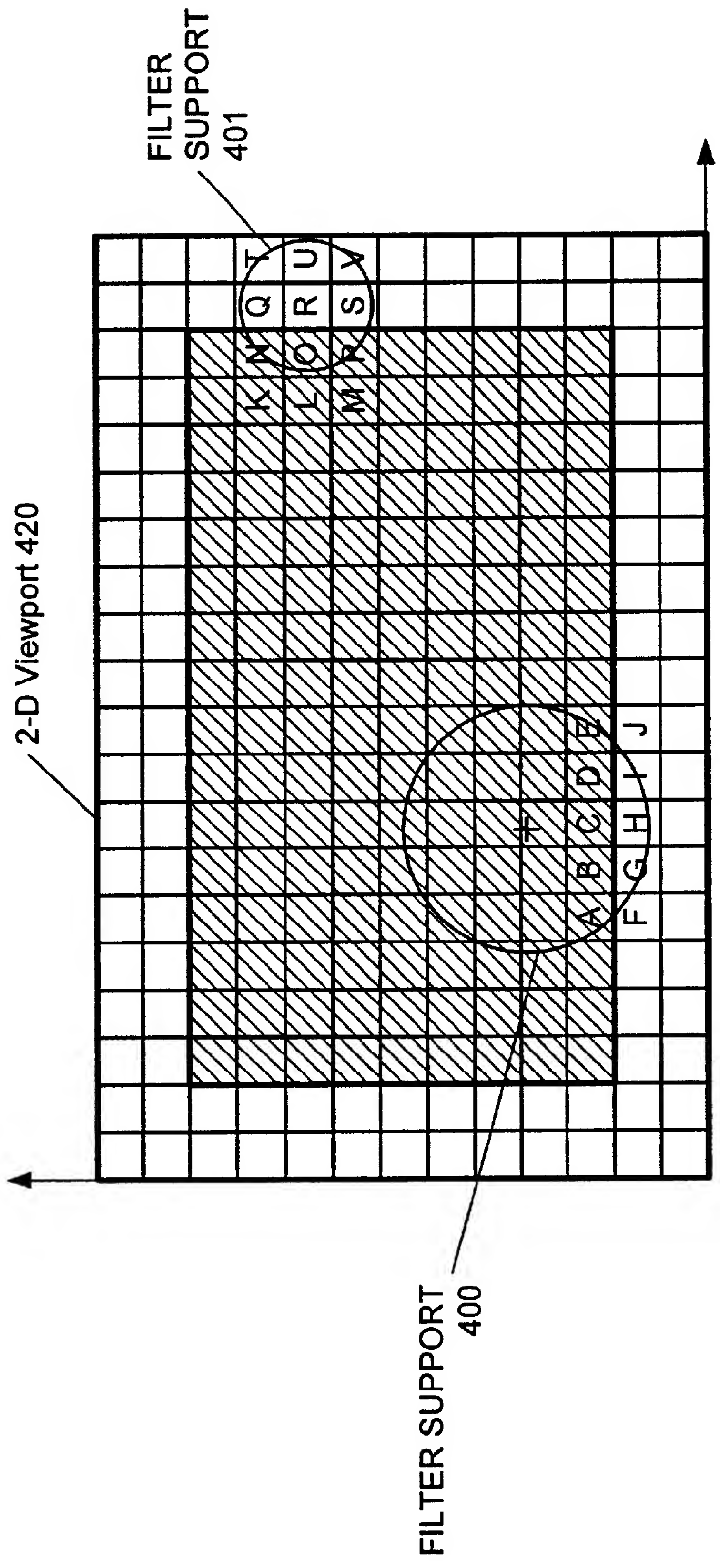
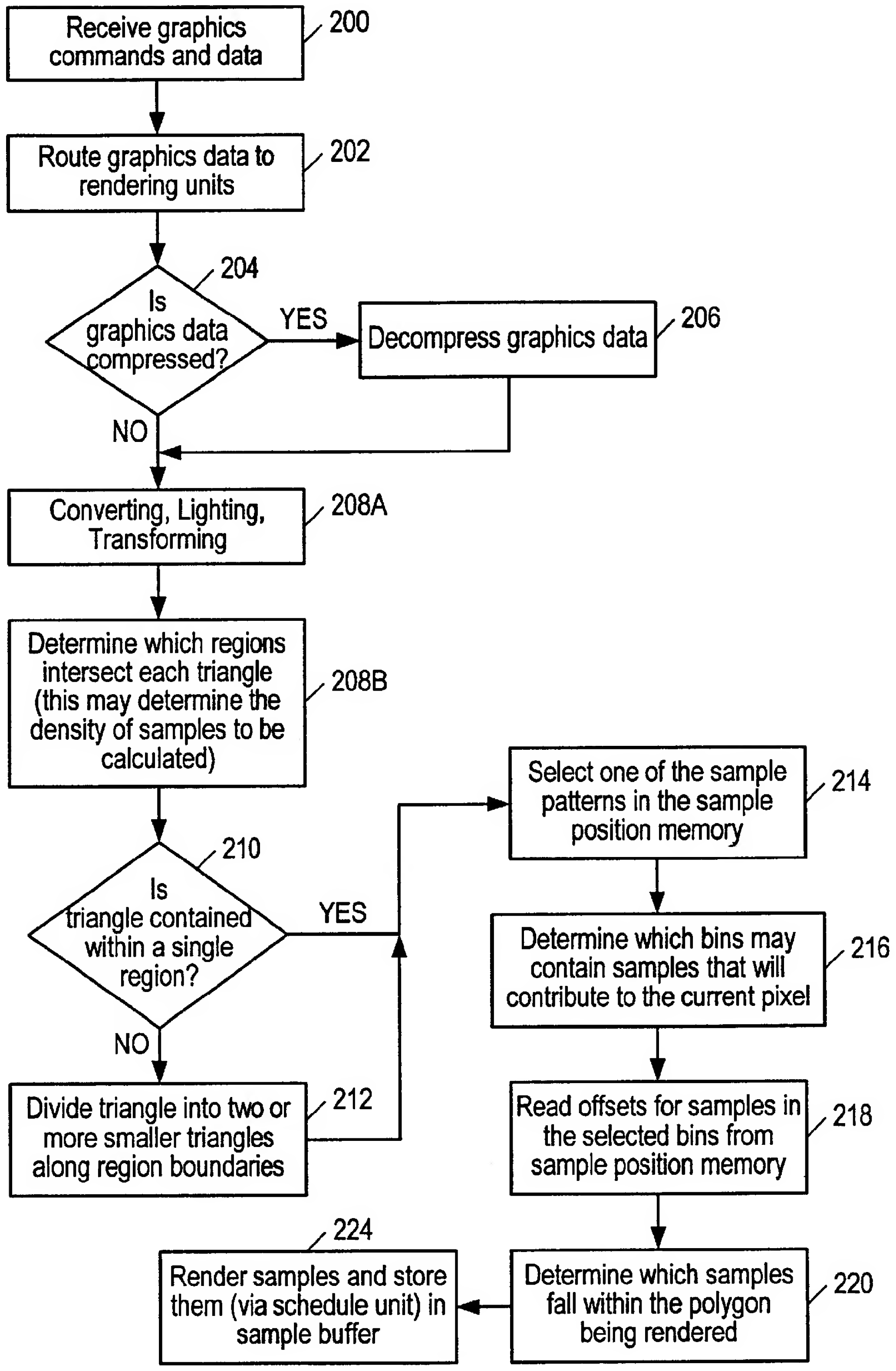

FILTER SUPPORT 401
2-D Viewport 420
FILTER SUPPORT 400
K N Q T
L O R U
M P S V
A B C D E
F G H I J

FIG. 16



Receive graphics commands and data
200
Route graphics data to rendering units
202
Is graphics data compressed?
204
YES
Decompress graphics data
206
NO
Converting, Lighting, Transforming
208A
Determine which regions intersect each triangle (this may determine the density of samples to be calculated)
208B
Is triangle contained within a single region?
210
YES
NO
Divide triangle into two or more smaller triangles along region boundaries
212
Select one of the sample patterns in the sample position memory
214
Determine which bins may contain samples that will contribute to the current pixel
216
Read offsets for samples in the selected bins from sample position memory
218
Determine which samples fall within the polygon being rendered
220
Render samples and store them (via schedule unit) in sample buffer
224

FIG. 17

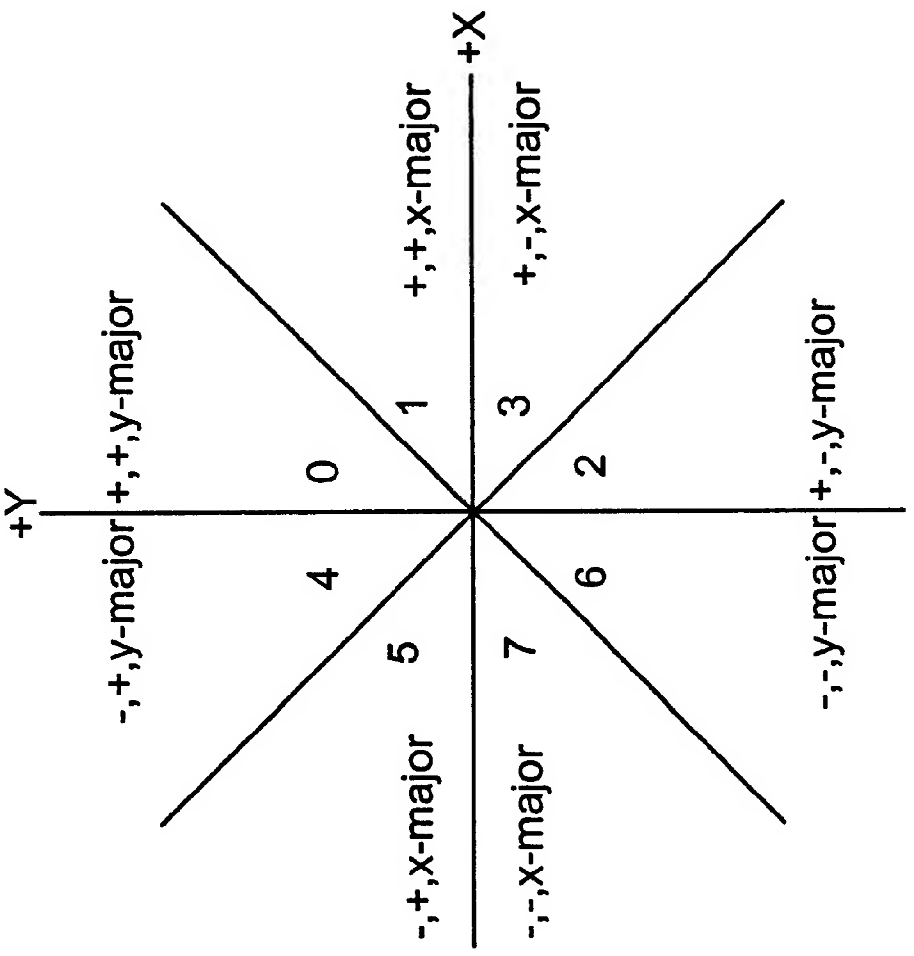

+X
+Y
+,+,x-major
+,-,x-major
+,+,y-major
-,+,y-major
+,-,y-major
-,-,y-major
-,+,x-major
-,-,x-major
0
1
2
3
4
5
6
7

FIG. 18

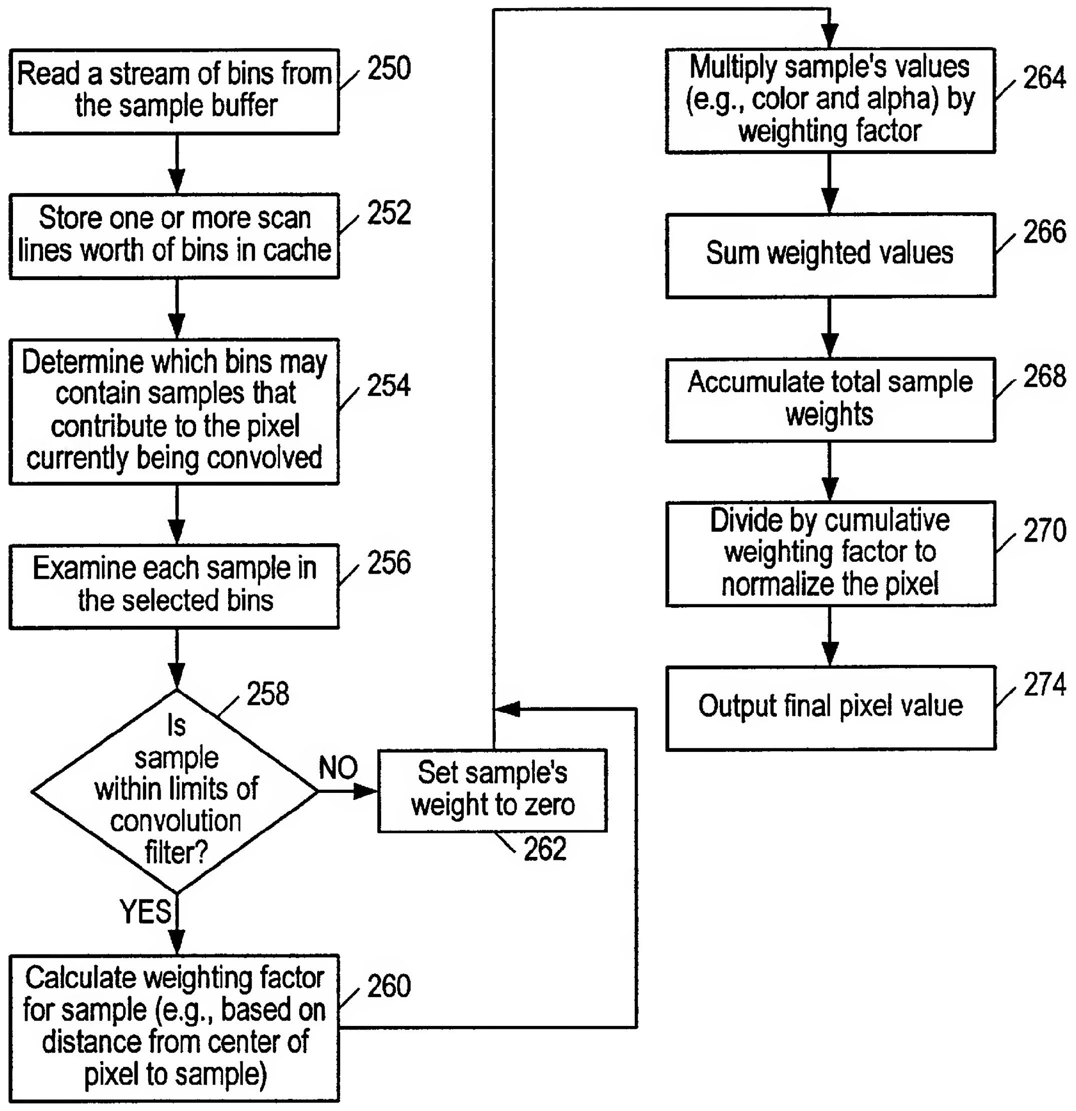
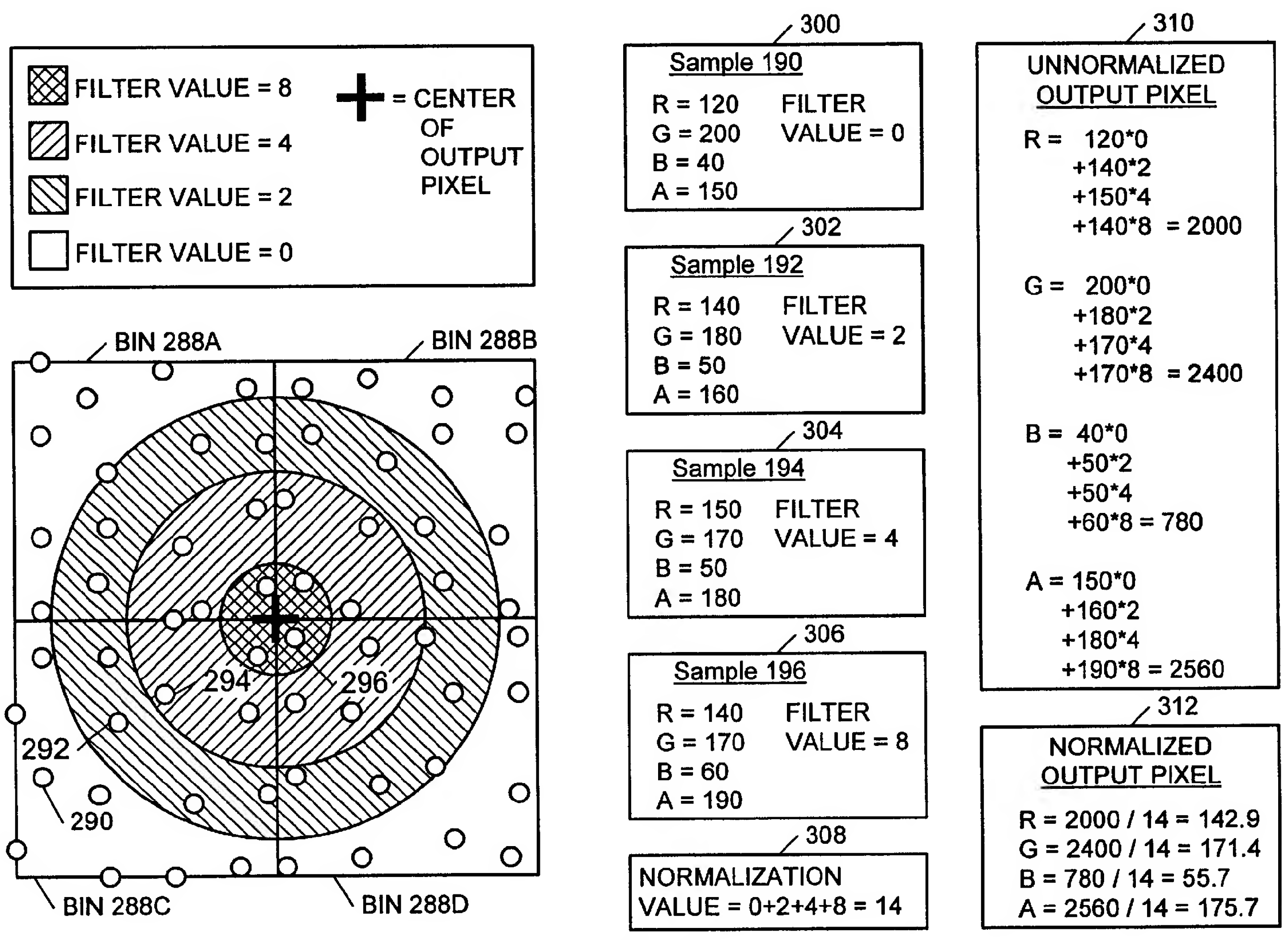

Read a stream of bins from the sample buffer
250
Store one or more scan lines worth of bins in cache
252
Determine which bins may contain samples that contribute to the pixel currently being convolved
254
Examine each sample in the selected bins
256
Is sample within limits of convolution filter?
258
NO
YES
Set sample's weight to zero
262
Calculate weighting factor for sample (e.g., based on distance from center of pixel to sample)
260
Multiply sample's values (e.g., color and alpha) by weighting factor
264
Sum weighted values
266
Accumulate total sample weights
268
Divide by cumulative weighting factor to normalize the pixel
270
Output final pixel value
274

FIG. 19



Sample 190
R = 120
G = 200
B = 40
A = 150
FILTER VALUE = 0

Sample 192
R = 140
G = 180
B = 50
A = 160
FILTER VALUE = 2

Sample 194
R = 150
G = 170
B = 50
A = 180
FILTER VALUE = 4

Sample 196
R = 140
G = 170
B = 60
A = 190
FILTER VALUE = 8

NORMALIZATION VALUE = 0+2+4+8 = 14

UNNORMALIZED OUTPUT PIXEL

R = 120*0 +140*2 +150*4 +140*8 = 2000

G = 200*0 +180*2 +170*4 +170*8 = 2400

B = 40*0 +50*2 +50*4 +60*8 = 780

A = 150*0 +160*2 +180*4 +190*8 = 2560

NORMALIZED OUTPUT PIXEL

R = 2000 / 14 = 142.9
G = 2400 / 14 = 171.4
B = 780 / 14 = 55.7
A = 2560 / 14 = 175.7

FIG. 20

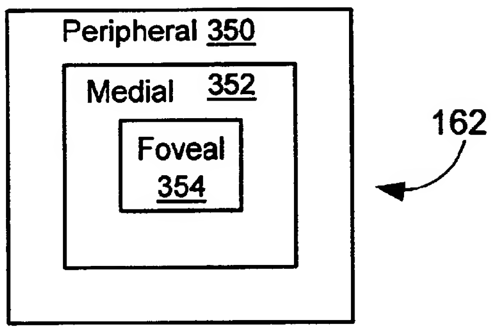


FIG. 21

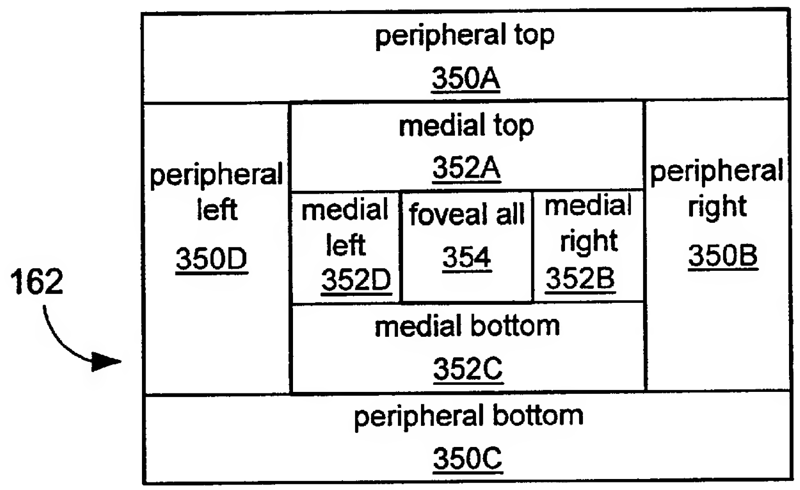


FIG. 22

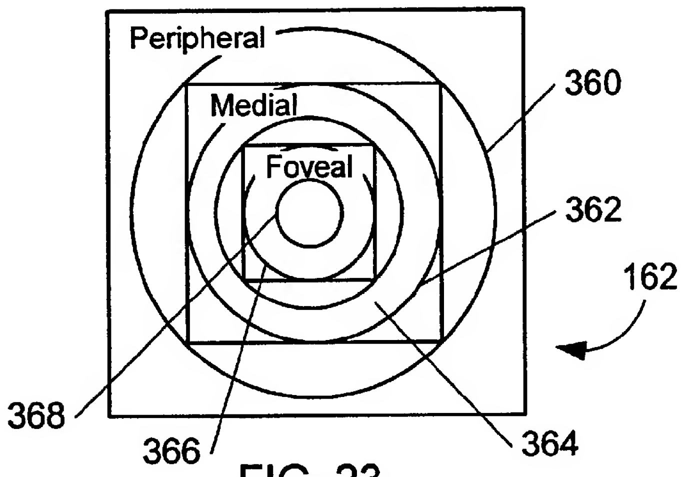
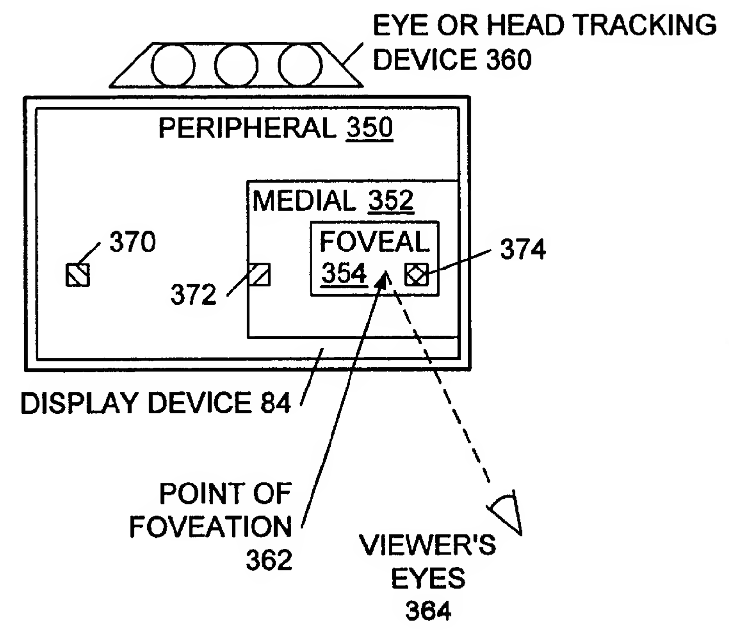


FIG. 23

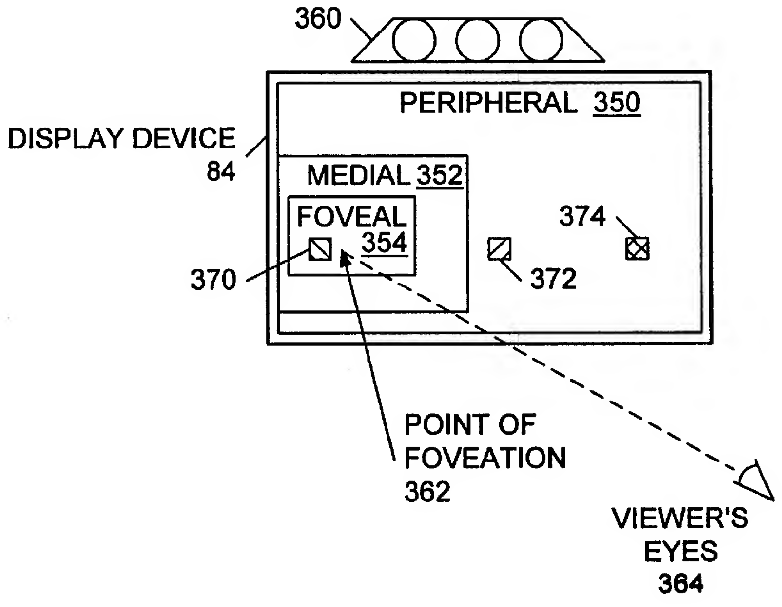


⊠ FOVEAL REGION = 8 SAMPLES PER BIN
CONVOLUTION RADIUS TOUCHES 4 BINS
TOTAL = 32 SAMPLES MAY CONTRIBUTE

⧄ MEDIAL REGION = 4 SAMPLES PER BIN
CONVOLUTION RADIUS TOUCHES 4 BINS
TOTAL = 16 SAMPLES MAY CONTRIBUTE

⧅ PERIPHERAL REGION = 1 SAMPLE PER BIN
CONVOLUTION RADIUS TOUCHES 1 BIN
TOTAL = 1 SAMPLE MAY CONTRIBUTE

FIG. 24A



⊠ PERIPHERAL REGION = 1 SAMPLE PER BIN
CONVOLUTION RADIUS TOUCHES 1 BIN
TOTAL = 1 SAMPLE MAY CONTRIBUTE

⧄ PERIPHERAL REGION = 1 SAMPLE PER BIN
CONVOLUTION RADIUS TOUCHES 1 BINS
TOTAL = 1 SAMPLE MAY CONTRIBUTE

⧅ FOVEAL REGION = 8 SAMPLES PER BIN
CONVOLUTION RADIUS TOUCHES 4 BIN
TOTAL = 32 SAMPLE MAY CONTRIBUTE

FIG. 24B

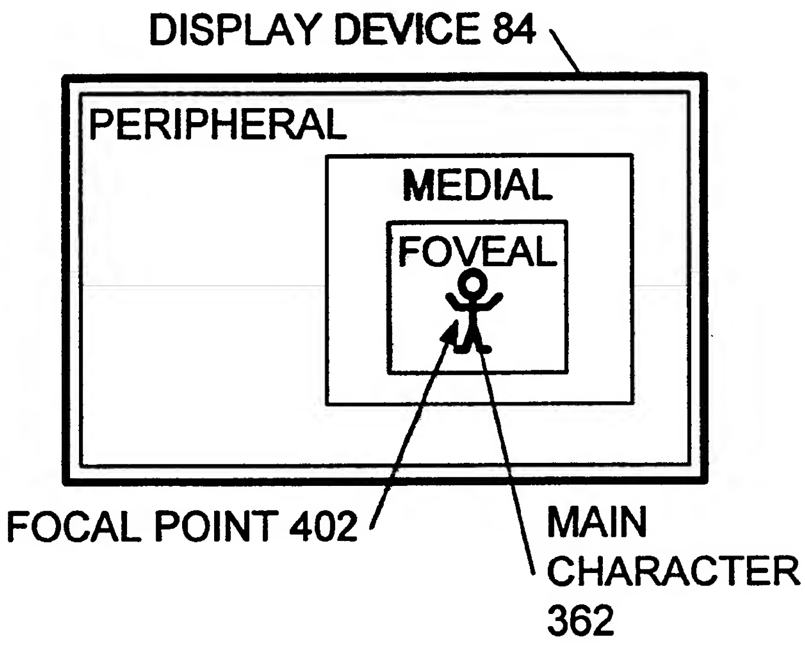


FIG. 25A

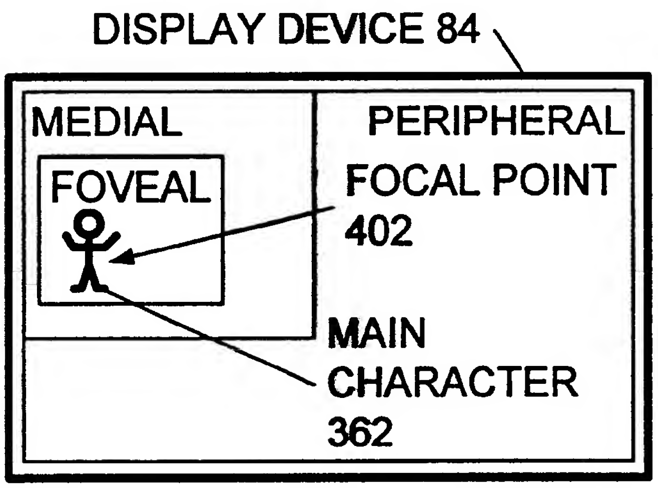


FIG. 25B

GRAPHICS SYSTEM CONFIGURED TO PERFORM PARALLEL SAMPLE TO PIXEL CALCULATION

This application is a continuation-in-part of co-pending application Ser. No. 09/251,844 titled "Graphics System With Programmable Real-Time Alpha Key Generation", filed on Feb. 17, 1999, which claims the benefit of U.S. Provisional Application No. 60/074,836, filed Feb. 17, 1998. These applications are hereby incorporated by reference in their entirety.

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates generally to the field of computer graphics and, more particularly, to high performance graphics systems.

2. Description of the Related Art

A computer system typically relies upon its graphics system for producing visual output on the computer screen or display device. Early graphics systems were only responsible for taking what the processor produced as output and displaying it on the screen. In essence, they acted as simple translators or interfaces. Modem graphics systems, however, incorporate graphics processors with a great deal of processing power. They now act more like coprocessors rather than simple translators. This change is due to the recent increase in both the complexity and amount of data being sent to the display device. For example, modem computer displays have many more pixels, greater color depth, and are able to display more complex images with higher refresh rates than earlier models. Similarly, the images displayed are now more complex and may involve advanced techniques such as anti-aliasing and texture mapping.

As a result, without considerable processing power in the graphics system, the CPU would spend a great deal of time performing graphics calculations. This could rob the computer system of the processing power needed for performing other tasks associated with program execution and thereby dramatically reduce overall system performance. With a powerful graphics system, however, when the CPU is instructed to draw a box on the screen, the CPU is freed from having to compute the position and color of each pixel. Instead, the CPU may send a request to the video card stating "draw a box at these coordinates." The graphics system then draws the box, freeing the processor to perform other tasks.

Generally, a graphics system in a computer (also referred to as a graphics system) is a type of video adapter that contains its own processor to boost performance levels. These processors are specialized for computing graphical transformations, so they tend to achieve better results than the general-purpose CPU used by the computer system. In addition, they free up the computer's CPU to execute other commands while the graphics system is handling graphics computations. The popularity of graphical applications, and especially multimedia applications, has made high performance graphics systems a common feature of computer systems. Most computer manufacturers now bundle a high performance graphics system with their systems.

Since graphics systems typically perform only a limited set of functions, they may be customized and therefore far more efficient at graphics operations than the computer's general-purpose central processor. While early graphics systems were limited to performing two-dimensional (2D) graphics, their functionality has increased to support three-dimensional (3D) wire-frame graphics, 3D solids, and now

includes support for three-dimensional (3D) graphics with textures and special effects such as advanced shading, fogging, alpha-blending, and specular highlighting.

The processing power of 3D graphics systems has been improving at a breakneck pace. A few years ago, shaded images of simple objects could only be rendered at a few frames per second, while today's systems support rendering of complex objects at 60 Hz or higher. At this rate of increase, in the not too distant future, graphics systems will literally be able to render more pixels than a single human's visual system can perceive. While this extra performance may be useable in multiple-viewer environments, it may be wasted in more common primarily single-viewer environments. Thus, a graphics system is desired which is capable of matching the variable nature of the human resolution system (i.e., capable of putting the quality where it is needed or most perceivable).

While the number of pixels is an important factor in determining graphics system performance, another factor of equal import is the quality of the image. For example, an image with a high pixel density may still appear unrealistic if edges within the image are too sharp or jagged (also referred to as "aliased"). One well-known technique to overcome these problems is anti-aliasing. Anti-aliasing involves smoothing the edges of objects by shading pixels along the borders of graphical elements. More specifically, anti-aliasing entails removing higher frequency components from an image before they cause disturbing visual artifacts. For example, anti-aliasing may soften or smooth high contrast edges in an image by forcing certain pixels to intermediate values (e.g., around the silhouette of a bright object superimposed against a dark background).

Another visual effect used to increase the realism of computer images is alpha blending. Alpha blending is a technique that controls the transparency of an object, allowing realistic rendering of translucent surfaces such as water or glass. Another effect used to improve realism is fogging. Fogging obscures an object as it moves away from the viewer. Simple fogging is a special case of alpha blending in which the degree of alpha changes with distance so that the object appears to vanish into a haze as the object moves away from the viewer. This simple fogging may also be referred to as "depth cueing" or atmospheric attenuation, i.e., lowering the contrast of an object so that it appears less prominent as it recedes. More complex types of fogging go beyond a simple linear function to provide more complex relationships between the level of translucence and an object's distance from the viewer. Current state of the art software systems go even further by utilizing atmospheric models to provide low-lying fog with improved realism.

While the techniques listed above may dramatically improve the appearance of computer graphics images, they also have certain limitations. In particular, they may introduce their own aberrations and are typically limited by the density of pixels displayed on the display device.

As a result, a graphics system is desired which is capable of utilizing increased performance levels to increase not only the number of pixels rendered but also the quality of the image rendered. In addition, a graphics system is desired which is capable of utilizing increases in processing power to improve the results of graphics effects such as anti-aliasing.

Prior art graphics systems have generally fallen short of these goals. Prior art graphics systems use a conventional frame buffer for refreshing pixel/video data on the display. The frame buffer stores rows and columns of pixels that exactly correspond to respective row and column locations on the display. Prior art graphics system render 2D and/or 3D images or objects into the frame buffer in pixel form, and then read the pixels from the frame buffer during a screen refresh to refresh the display. Thus, the frame buffer stores the output pixels that are provided to the display. To reduce visual artifacts that may be created by refreshing the screen at the same time the frame buffer is being updated, most graphics systems' frame buffers are double-buffered.

To obtain more realistic images, some prior art graphics systems have gone further by generating more than one sample per pixel. As used herein, the term "sample" refers to calculated color information that indicates the color, depth (z), transparency, and potentially other information, of a particular point on an object or image. For example a sample may comprise the following component values: a red value, a green value, a blue value, a z value, and an alpha value (e.g., representing the transparency of the sample). A sample may also comprise other information, e.g., a z-depth value, a blur value, an intensity value, brighter-than-bright information, and an indicator that the sample consists partially or completely of control information rather than color information (i.e., "sample control information"). By calculating more samples than pixels (i.e., super-sampling), a more detailed image is calculated than can be displayed on the display device. For example, a graphics system may calculate four samples for each pixel to be output to the display device. After the samples are calculated, they are then combined or filtered to form the pixels that are stored in the frame buffer and then conveyed to the display device. Using pixels formed in this manner may create a more realistic final image because overly abrupt changes in the image may be smoothed by the filtering process.

These prior art super-sampling systems typically generate a number of samples that are far greater than the number of pixel locations on the display. These prior art systems typically have rendering processors that calculate the samples and store them into a render buffer. Filtering hardware then reads the samples from the render buffer, filters the samples to create pixels, and then stores the pixels in a traditional frame buffer. The traditional frame buffer is typically double-buffered, with one side being used for refreshing the display device while the other side is updated by the filtering hardware. Once the samples have been filtered, the resulting pixels are stored in a traditional frame buffer that is used to refresh to display device. These systems, however, have generally suffered from limitations imposed by the conventional frame buffer and by the added latency caused by the render buffer and filtering. Therefore, an improved graphics system is desired which includes the benefits of pixel super-sampling while avoiding the drawbacks of the conventional frame buffer.

U.S. patent application Ser. No. 09/251,844 titled "Graphics System with a Variable Resolution Sample Buffer" discloses a computer graphics system that utilizes a super-sampled sample buffer and a sample-to-pixel calculation unit for refreshing the display. The graphics processor generates a plurality of samples and stores them into a sample buffer. The graphics processor preferably generates and stores more than one sample for at least a subset of the pixel locations on the display. Thus, the sample buffer is a super-sampled sample buffer which stores a number of samples that may be far greater than the number of pixel locations on the display. The sample-to-pixel calculation unit is configured to read the samples from the super-sampled sample buffer and filter or convolve the samples into respective output pixels, wherein the output pixels are

then provided to refresh the display. The sample-to-pixel calculation unit selects one or more samples and filters them to generate an output pixel. The sample-to-pixel calculation unit may operate to obtain samples and generate pixels which are provided directly to the display with no frame buffer there between.

SUMMARY OF THE INVENTION

The problems set forth above may at least in part be solved by a graphics system that is configured to utilize a sample buffer and a plurality of parallel sample-to-pixel calculation units, wherein the sample-pixel calculation units are configured to access different portions of the sample buffer in parallel. Advantageously, this configuration (depending upon the embodiment) may also allow the graphics system to use a sample buffer in lieu of a traditional frame buffer that stores pixels. Since the sample-to-pixel calculation units may be configured to operate in parallel, the latency of the graphics system may be reduced in some embodiments.

In one embodiment, the graphics system may include one or more graphics processors, a sample buffer, and a plurality of sample-to-pixel calculation units. The graphics processors may be configured to receive a set of three-dimensional graphics data and render a plurality of samples based on the graphics data. The sample buffer may be configured to store the plurality of samples (e.g., in a double-buffered configuration) for the sample-to-pixel calculation units, which are configured to receive and filter samples from the sample buffer to create output pixels. The output pixels are usable to form an image on a display device. Each of the sample-to-pixel calculation units are configured to generate pixels corresponding to a different region of the image. The region may be a vertical stripe (i.e., a column) of the image, a horizontal stripe (i.e., a row) of the image, or a rectangular portion of the image. Note, as used herein the terms "horizontal row" and "horizontal stripe" are used interchangeably, as are "vertical column" and "vertical stripe". Each region may overlap the other regions of the image to prevent visual aberrations (e.g., seams, lines, or tears in the image). As previously noted, each of the sample-to-pixel calculation units may advantageously be configured to operate in parallel on its own region or regions. The sample-to-pixel calculation units are configured to process the samples by (i) determining which samples are within a predetermined filter envelope, (ii) multiplying those samples by a weighting, (iii) summing the resulting values, and (iv) normalizing the results to form output pixels. The weighting value may vary with respect the sample's position within the filter envelope (e.g., the weighting factor may decrease as the samples move farther from the center of the filter envelope). In some embodiments, the weighting factor may be normalized or pre-normalized, in which case the resulting output pixel will not proceed through normalization because the output will already be normalized. Normalized weighting factors are adjusted to ensure that pixels generated with fewer contributing samples will not overpower pixels generated with more contributing samples. In contrast, if un-normalized weighting factors are used, the resulting pixel will typically proceed through normalization. Normalization will typically be performed in embodiments of the graphics system that allow for a variable number of samples to contribute to each output pixel. Normalization may also be performed in systems that allow variable sample patterns, and in systems in which the pitch of the centers of filters vary widely with respect to the sample pattern.

In some embodiments, the graphics system may be configured to dynamically change the size or type of regions being used (e.g., changing the width of the vertical columns used on a frame-by-frame basis). Some embodiments of the graphics system may support a variable resolution or variable density frame buffer. In these configurations, the graphics system is configured to render samples more densely in certain areas of the image (e.g., the center of the image or the portion of the image where the viewer's attention is most likely focused). Advantageously, the ability to dynamically vary the size and/or shape of the regions used may allow the graphics system to equalize (or come closer to equalizing) the number of samples that each sample-to-pixel calculation unit processes for a particular frame.

The samples may include color components and alpha (e.g., transparency) components, and may be stored in "bins" to simplify the process of storing and retrieving samples from the sample buffer. As described in greater detail below, bins are a means for organizing and dividing the sample buffer into smaller sets of storage locations. In addition, in some embodiments the three-dimensional graphics data may be received in a compressed form (e.g., using geometry compression). In these embodiments the graphics processors may be configured to decompress the three-dimensional graphics data before rendering the samples. As used herein, the term "color components" includes information on a per-sample or per-pixel basis that is usable to determine the color the pixel or sample. For example, RGB information and transparency information may be color components.

A method for rendering a set of three-dimensional graphics data is also contemplated. In one embodiment the method comprises: (i) receiving the three-dimensional graphics data, (ii) generating one or more samples based on the graphics data, (iii) storing the samples, (iv) selecting stored samples; and (iv) filtering the selected samples in parallel to form output pixels. The stored samples may be selected according to a plurality of regions, as described above.

BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing, as well as other objects, features, and advantages of this invention may be more completely understood by reference to the following detailed description when read together with the accompanying drawings in which:

FIG. **1**A illustrates one embodiment of a computer system that includes one embodiment of a graphics system;

FIG. **1**B illustrates another embodiment of a computer system that is part of a virtual reality work station;

FIG. **2** illustrates one embodiment of a network to which the computers systems of FIGS. **1**A–B may be connected;

FIG. **3**A is a diagram illustrating another embodiment of the graphics system of FIG. **1** as a virtual reality work station;

FIG. **3**B is more detailed diagram illustrating one embodiment of a graphics system with a sample buffer;

FIG. **4** illustrates traditional pixel calculation;

FIG. **5**A illustrates one embodiment of super-sampling;

FIG. **5**B illustrates a random distribution of samples;

FIG. **6** illustrates details of one embodiment of a graphics system having one embodiment of a variable resolution super-sampled sample buffer;

FIG. **7** illustrates details of another embodiment of a graphics system having one embodiment of a variable resolution super-sampled sample buffer and a double buffered sample position memory;

FIG. **8** illustrates details of three different embodiments of sample positioning schemes;

FIG. **9** illustrates details of one embodiment of a sample positioning scheme;

FIG. **10** illustrates details of another embodiment of a sample positioning scheme;

FIG. **11**A illustrates details of one embodiment of a graphics system configured to convert samples to pixels in parallel using vertical screen stripes (columns);

FIG. **11**B illustrates details of another embodiment of a graphics system configured to convert samples to pixels in parallel using vertical screen stripes (columns);

FIG. **12** illustrates details of another embodiment of a graphics system configured to convert samples to pixels in parallel using horizontal screen stripes (rows);

FIG. **13** illustrates details of another embodiment of a graphics system configured to convert samples to pixels in parallel using rectangular regions;

FIG. **14** illustrates details of one method for reading samples from a sample buffer;

FIG. **15** illustrates details of one embodiment of a method for dealing with boundary conditions;

FIG. **16** illustrates details of another embodiment of a method for dealing with boundary conditions;

FIG. **17** is a flowchart illustrating one embodiment of a method for drawing samples into a super-sampled sample buffer;

FIG. **18** illustrates one embodiment of a method for coding triangle vertices;

FIG. **19** illustrates one embodiment of a method for calculating pixels from samples;

FIG. **20** illustrates details of one embodiment of a sample to pixel calculation for an example set of samples;

FIG. **21** illustrates one embodiment of a method for varying the density of samples;

FIG. **22** illustrates another embodiment of a method for varying the density of samples;

FIG. **23** illustrates yet another embodiment of a method for varying the density of samples;

FIGS. **24**A–B illustrate details of one embodiment of a method for utilizing eye-tracking to vary the density of samples; and

FIGS. **25**A–B illustrate details of one embodiment of a method for utilizing eye-tracking to vary the density of samples.

While the invention is susceptible to various modifications and alternative forms, specific embodiments thereof are shown by way of example in the drawings and will herein be described in detail. It should be understood, however, that the drawings and detailed description thereto are not intended to limit the invention to the particular form disclosed, but on the contrary, the intention is to cover all modifications, equivalents, and alternatives falling within the spirit and scope of the present invention as defined by the appended claims.

DETAILED DESCRIPTION OF SEVERAL EMBODIMENTS

Computer System—FIG. 1A

Referring now to FIG. **1**A, one embodiment of a computer system **80** that includes a three-dimensional (3-D) graphics system is shown. The 3-D graphics system may be comprised in any of various systems, including a computer system, network PC, Internet appliance, a television, including HDTV systems and interactive television systems, personal digital assistants (PDAs), and other devices which display 2D and/or 3D graphics, among others.

As shown, the computer system **80** comprises a system unit **82** and a video monitor or display device **84** coupled to the system unit **82**. The display device **84** may be any of various types of display monitors or devices (e.g., a CRT, LCD, or gas-plasma display). Various input devices may be connected to the computer system, including a keyboard **86** and/or a mouse **88**, or other input device (e.g., a trackball, digitizer, tablet, six-degree of freedom input device, head tracker, eye tracker, data glove, body sensors, etc.). Application software may be executed by the computer system **80** to display 3-D graphical objects on display device **84**. As described further below, the 3-D graphics system in computer system **80** includes a super-sampled sample buffer with a programmable real-time sample-to-pixel calculation unit to improve the quality and realism of images displayed on display device **84**.

Computer system **80** may also include eye-tracking sensor **92** and/or 3D-glasses **90**. 3D glasses **90** may be active (e.g., LCD shutter-type) or passive (e.g., polarized, red-green, etc.) and may allow the user to view a more three-dimensional image on display device **84**. With glasses **90**, each eye receives a slightly different image, which the viewer's mind interprets as a "true" three-dimensional view. Sensor **92** may be configured to determine which part of the image on display device **84** that the viewer is looking at (i.e., that the viewer's field of view is centered on). The information provided by sensor **92** may used in a number of different ways as will be described below.

Virtual Reality Computer System—FIG. 1B

FIG. **1B** illustrates another embodiment of a computer system **70**. In this embodiment, the system comprises a head-mounted display device **72**, head-tracking sensors **74**, and a data glove **76**. Head mounted display **72** may be coupled to system unit **82** via a fiber optic link **94**, or one or more of the following: an electrically-conductive link, an infra-red link, or a wireless (e.g., RF) link. Other embodiments are possible and contemplated.

Computer Network—FIG. 2

Referring now to FIG. **2**, a computer network **500** is shown comprising at least one server computer **502** and one or more client computers **506**A–N. (In the embodiment shown in FIG. **4**, client computers **506**A–B are depicted). One or more of the client systems may be configured similarly to computer system **80**, with each having one or more graphics systems **112** as described above. Server **502** and client(s) **506** may be joined through a variety of connections **504**, such as a local-area network (LAN), a wide-area network (WAN), or an Internet connection. In one embodiment, server **502** may store and transmit 3-D geometry data (which may be compressed) to one or more of clients **506**. The clients **506** receive the compressed 3-D geometry data, decompress it (if necessary) and then render the geometry data. The rendered image is then displayed on the client's display device. The clients render the geometry data and display the image using super-sampled sample buffer and real-time filter techniques described above. In another embodiment, the compressed 3-D geometry data may be transferred between client computers **506**.

Computer System Block Diagram—FIG. 3A

FIG. **3A** presents a simplified block diagram for computer system **80**. Elements of computer system **80** that are not

necessary for an understanding of the present invention are suppressed for convenience. Computer system **80** comprises a host central processing unit (CPU) **102** and a 3-D graphics system **112** coupled to system bus **104**. A system memory **106** may also be coupled to system bus **104**.

Host CPU **102** may be realized by any of a variety of processor technologies. For example, host CPU **102** may comprise one or more general purpose microprocessors, parallel processors, vector processors, digital signal processors, etc., or any combination thereof. System memory **106** may include one or more memory subsystems representing different types of memory technology. For example, system memory **106** may include read-only memory (ROM), random access memory (RAM)—such as static random access memory (SRAM), synchronous dynamic random access memory (SDRAM), and Rambus dynamic random access memory (RDRAM)—and mass storage devices.

System bus **104** may comprise one or more communication buses or host computer buses (for communication between host processors and memory subsystems). In addition, various peripheral devices and peripheral buses may be connected to system bus **104**.

Graphics system **112** is configured according to the principles of the present invention, and may couple to system bus **104** by a crossbar switch or any other type of bus connectivity logic. Graphics system **112** drives each of projection devices PD_I–PD_L and display device **84** with a corresponding video signal.

It is noted that the 3-D graphics system **112** may couple to one or more busses of various types in addition to system bus **104**. Furthermore, the 3D graphics system **112** may couple to a communication port, and thereby, directly receive graphics data from an external source such as the Internet or a local area network.

Host CPU **102** may transfer information to/from graphics system **112** according to a programmed input/output (I/O) protocol over system bus **104**. Alternately, graphics system **112** may access system memory **106** according to a direct memory access (DMA) protocol or through intelligent bus-mastering.

A graphics application program conforming to an application programming interface (API) such as OpenGL® (a registered trademark of Silicon Graphics, Inc.) or Java3D™ (a trademark of Sun Microsystems, Inc.) may execute on host CPU **102** and generate commands and data that define a geometric primitive such as a polygon for output on projection devices PD_I through PD_L and/or display device **84**. Host CPU **102** may transfer this graphics data to system memory **106**. Thereafter, the host CPU **102** may transfer the graphics data to graphics system **112** over system bus **104**. In another embodiment, graphics system **112** may read geometry data arrays from system memory **106** using DMA access cycles. In yet another embodiment, graphics system **112** may be coupled to system memory **106** through a direct port, such as an Advanced Graphics Port (AGP) promulgated by Intel Corporation.

Graphics system **112** may receive graphics data from any of various sources including host CPU **102**, system memory **106** or any other memory, external sources such as a network (e.g., the Internet) or a broadcast medium (e.g. television).

As will be described below, graphics system **112** may be configured to allow more efficient microcode control, which results in increased performance for handling of incoming color values corresponding to the polygons generated by host CPU **102**.

While graphics system **112** is depicted as part of computer system **80**, graphics system **112** may also be configured as a stand-alone device. Graphics system **112** may also be configured as a single chip device or as part of a system-on-a-chip or a multi-chip module.

Graphics system **112** may be comprised in any of various systems, including a network PC, an Internet appliance, a television (including an HDTV system or an interactive television system), a personal digital assistant (PDA), or other devices which display 2D and/or 3D graphics.

As described further below, the 3-D graphics system within in computer system **80** includes a super-sampled sample buffer and a plurality of programmable sample-to-pixel calculation units to improve the quality and realism of images displayed by projection devices PD_1 through PD_L and/or display device **84**. Each sample-to-pixel calculation unit may include a filter (i.e., convolution) pipeline or other hardware for generating pixel values (e.g. red, green and blue values) based on samples in the sample buffer. Each sample-to-pixel calculation unit may obtain samples from the sample buffer and generate pixel values which are provided to any of projection devices PD_1 through PD_L or display device **84**. The sample-to-pixel calculation units may operate in a "real-time" or "on-the-fly" fashion.

As used herein the terms "filter" and "convolve" are used interchangeably. As used herein, the term "real-time" refers to a process or operation that is performed at or near the refresh rate of projection devices PD_1 through PD_L or display device **84**. The term "on-the-fly" refers to a process or operation that generates images at a rate near or above the minimum rate required for displayed motion to appear smooth (i.e., motion fusion) and for the light intensity to appear continuous (i.e., flicker fusion). These concepts are further described in the book "Spatial Vision" by Russel L. De Valois and Karen K. De Valois, Oxford University Press, 1988.

Graphics System—FIG. 3B

FIG. **3B** presents a block diagram for one embodiment of graphics system **112** according to the present invention. Graphics system **112** may comprise a graphics processing unit (GPU) **90**, one or more super-sampled sample buffers **162**, and one or more sample-to-pixel calculation units **170-1** through **170**-V. Graphics system **112** may also comprise one or more digital-to-analog converters (DACs) **178-1** through **178**-L. Graphics processing unit **90** may comprise any combination of processor technologies. For example, graphics processing unit **90** may comprise specialized graphics processors or calculation units, multimedia processors, DSPs, or general purpose processors.

In one embodiment, graphics processing unit **90** may comprise one or more rendering units **150**A–D. Graphics processing unit **90** may also comprise one or more control units **140**, one or more data memories **152**A–D, and one or more schedule units **154**. Sample buffer **162** may comprise one or more sample memories **160**A–**160**N.

A. Control Unit 140

Control unit **140** operates as the interface between graphics system **112** and computer system **80** by controlling the transfer of data between graphics system **112** and computer system **80**. In embodiments of graphics system **112** that comprise two or more rendering units **150**A–D, control unit **140** may also divide the stream of data received from computer system **80** into a corresponding number of parallel streams that are routed to the individual rendering units **150**A–D. The graphics data may be received from computer system **80** in a compressed form. Graphics data compression may advantageously reduce the required transfer bandwidth between computer system **80** and graphics system **112**. In one embodiment, control unit **140** may be configured to split and route the received data stream to rendering units **150**A–D in compressed form.

The graphics data may comprise one or more graphics primitives. As used herein, the term graphics primitive includes polygons, parametric surfaces, splines, NURBS (non-uniform rational B-splines), subdivision surfaces, fractals, volume primitives, and particle systems. These graphics primitives are described in detail in the text book entitled "Computer Graphics: Principles and Practice" by James D. Foley, et al., published by Addison-Wesley Publishing Co., Inc., 1996.

It is noted that the embodiments and examples of the invention presented herein are described in terms of polygons for the sake of simplicity. However, any type of graphics primitive may be used instead of or in addition to polygons in these embodiments and examples.

B. Rendering Units

Rendering units **150**A–D (also referred to herein as draw units) are configured to receive graphics instructions and data from control unit **140** and then perform a number of functions which depend on the exact implementation. For example, rendering units **150**A–D may be configured to perform decompression (if the received graphics data is presented in compressed form), transformation, clipping, lighting, texturing, depth cueing, transparency processing, set-up, visible object determination, and virtual screen rendering of various graphics primitives occurring within the graphics data.

Depending upon the type of compressed graphics data received, rendering units **150**A–D may be configured to perform arithmetic decoding, run-length decoding, Huffman decoding, and dictionary decoding (e.g., LZ**77**, LZSS, LZ**78**, and LZW). In another embodiment, rendering units **150**A–D may be configured to decode graphics data that has been compressed using geometric compression. Geometric compression of 3D graphics data may achieve significant reductions in data size while retaining most of the image quality. Two methods for compressing and decompressing 3D geometry are described in:

U.S. Pat. No. 5,793,371, application Ser. No. 08/511,294, filed on Aug. 4, 1995, entitled "Method And Apparatus For Geometric Compression Of Three-Dimensional Graphics Data," Attorney Docket No. 5181-05900; and

U.S. patent application Ser. No. 09/095,777, filed on Jun. 11, 1998, entitled "Compression of Three-Dimensional Geometry Data Representing a Regularly Tiled Surface Portion of a Graphical Object," Attorney Docket No. 5181-06602.

In embodiments of graphics system **112** that support decompression, the graphics data received by each rendering unit **150** is decompressed into one or more graphics "primitives" which may then be rendered. The term primitive refers to components of objects that define its shape (e.g., points, lines, triangles, polygons in two or three dimensions, polyhedra, voxels, or free-form surfaces in three dimensions). Each rendering unit **150** may be any suitable type of high performance processor (e.g., a specialized graphics processor or calculation unit, a multimedia processor, a digital signal processor, or a general purpose processor).

Transformation refers to applying a geometric operation to a primitive or an object comprising a set of primitives. For example, an object represented by a set of vertices in a local coordinate system may be embedded with arbitrary position, orientation, and size in world space using an appropriate sequence of translation, rotation, and scaling transformations. Transformation may also comprise reflection, skewing, or any other affine transformation. More generally, transformations may comprise nonlinear operations.

Lighting refers to calculating the illumination of objects. Lighting computations result in an assignment of color and/or brightness to objects or to selected points (e.g. vertices) on objects. Depending upon the shading algorithm being used (e.g., constant, Gourand, or Phong shading), lighting may be evaluated at a number of different locations. For example, if constant shading is used (i.e., the lighted surface of a polygon is assigned a constant illumination value), then the lighting need only be calculated once per polygon. If Gourand shading is used, then the lighting is calculated once per vertex. Phong shading calculates the lighting on a per-sample basis.

Clipping refers to the elimination of graphics primitives or portions of graphics primitives which lie outside of a 3-D view volume in world space. The 3-D view volume may represent that portion of world space which is visible to a virtual observer situated in world space. For example, the view volume may be a solid cone generated by a 2-D view window and a view point located in world space. The solid cone may be imagined as the union of all rays emanating from the view point and passing through the view window. The view point may represent the world space location of the virtual observer. Primitives or portions of primitives which lie outside the 3-D view volume are not currently visible and may be eliminated from further processing. Primitives or portions of primitives which lie inside the 3-D view volume are candidates for projection onto the 2-D view window.

In order to simplify the clipping and projection computations, primitives may be transformed into a second, more convenient, coordinate system referred to herein as the viewport coordinate system. In viewport coordinates, the view volume maps to a canonical 3-D viewport which may be more convenient for clipping against. The term set-up refers to this mapping of graphics primitives into viewport coordinates.

Graphics primitives or portions of primitives which survive the clipping computation may be projected onto a 2-D viewport depending on the results of a visibility determination. Instead of clipping in 3-D, graphics primitives may be projected onto a 2-D view plane (which includes the 2-D viewport) and then clipped with respect to the 2-D viewport.

Virtual display rendering refers to calculations that are performed to generate samples for projected graphics primitives. For example, the vertices of a triangle in 3-D may be projected onto the 2-D viewport. The projected triangle may be populated with samples, and values (e.g. red, green, blue and z values) may be assigned to the samples based on the corresponding values already determined for the projected vertices. For example, the red value for each sample in the projected triangle may be interpolated from the known red values of the vertices. These sample values for the projected triangle may be stored in sample buffer **162**. Depending upon the embodiment, sample buffer **16** also stores a z value for each sample. This z-value is stored with the sample for a number of reasons, including depth-buffering. As samples for successive primitives are rendered, a virtual image accumulates in sample buffer **162**. Thus, the 2-D viewport is said to be a virtual screen on which the virtual image is rendered. The sample values comprising the virtual image are stored into sample buffer **162**. Points in the 2-D viewport are described in terms of virtual screen coordinates X and Y, and are said to reside in virtual screen space.

When the virtual image is complete, e.g., when all graphics primitives have been rendered, sample-to-pixel calculation units **170** may access the samples comprising the virtual image and may filter the samples to generate pixel values. In other words, the sample-to-pixel calculation units **170** may perform a spatial convolution of the virtual image with respect to a convolution kernel f(X,Y) to generate pixel values. For example, a red value R_p for a pixel P may be computed at any location (X_p,Y_p) in virtual screen space based on the relation

$$R_p = \frac{1}{E}\sum f(X_k - X_p, Y_k - Y_p)R(X_k, Y_k),$$

where the summation is evaluated at samples (X_k,Y_k) in the neighborhood of location (X_p, Y_p). Since convolution kernel f(X,Y) is non-zero only in a neighborhood of the origin, the displaced kernel $f(X-X_p, Y-Y_p)$ may take non-zero values only in a neighborhood of location (X_p,Y_p). The value E is a normalization value that may be computed according to the relation

$$E=\Sigma f(X_k-X_p,Y_k-Y_p),$$

where the summation is evaluated in the same neighborhood as above. The summation for the normalization value E may be performed in parallel with the summation for the red pixel value R_p. The location (X_p,Y_p) may be referred to as a pixel center or pixel origin. In the case where the convolution kernel f(X,Y) is symmetric with respect to the origin (0,0), the term pixel center maybe used.

The pixel values may be presented to projection devices PD_l through PD_L for display on projection screen SCR. The projection devices each generate a portion of integrated image IMG. Sample-to-pixel calculation units **170** may also generate pixel values for display on display device **84**.

In the embodiment of graphics system **112** shown in FIG. **3**, rendering units **150**A–D calculate sample values instead of pixel values. This allows rendering units **150**A–D to perform super-sampling, i.e. to calculate more than one sample per pixel. Super-sampling in the context of the present invention is discussed more thoroughly below. More details on super-sampling are discussed in the following books: "Principles of Digital Image Synthesis" by Andrew Glassner, 1995, Morgan Kaufman Publishing (Volume 1); and "Renderman Companion:" by Steve Upstill, 1990, Addison Wesley Publishing.

Sample buffer **162** may be double-buffered so that rendering units **150**A–D may write samples for a first virtual image into a first portion of sample buffer **162**, while a second virtual image is simultaneously read from a second portion of sample buffer **162** by sample-to-pixel calculations units **170**.

It is noted that the 2-D viewport and the virtual image which is rendered with samples into sample buffer **162** may correspond to an area larger than that area which is physically displayed as integrated image IMG or display image DIM. For example, the 2-D viewport may include a viewable subwindow. The viewable subwindow may correspond to integrated image IMG and/or display image DIM, while the marginal area of the 2-D viewport (outside the viewable subwindow) may allow for various effects such as panning and zooming. In other words, only that portion of the virtual image which lies within the viewable subwindow gets physically displayed. In one embodiment, the viewable subwindow equals the whole of the 2-D viewport. In this case, all of the virtual image gets physically displayed.

Note that rendering units **150**A–D may comprise a number of smaller and more specialized functional units, e.g., one or more set-up/decompress units and one or more lighting units.

C. Data Memories

Each of rendering units **150**A–D may be coupled to a corresponding one of instruction and data memories **152**A–D. In one embodiment, each of memories **152**A–D may be configured to store both data and instructions for a corresponding one of rendering units **150**A–D. While implementations may vary, in one embodiment, each data memory **152**A–D may comprise two 8 MByte SDRAMs, providing a total of 16 MBytes of storage for each rendering unit **150**A–D. In another embodiment, RDRAMs (Rambus DRAMs) may be used to support the decompression and set-up operations of each rendering unit, while SDRAMs may be used to support the draw functions of each rendering unit. Data memories **152**A–D may also be referred to as texture and render memories **152**A–D.

D. Schedule Unit

Schedule unit **154** may be coupled between rendering units **150**A–D and sample memories **160**A–N. Schedule unit **154** is configured to sequence the completed samples and store them in sample memories **160**A–N. Note in larger configurations, multiple schedule units **154** may be used in parallel. In one embodiment, schedule unit **154** may be implemented as a crossbar switch.

E. Sample Memories

Super-sampled sample buffer **162** comprises sample memories **160A**–**160N**, which are configured to store the plurality of samples generated by rendering units **150**A–D. As used herein, the term "sample buffer" refers to one or more memories which store samples. As previously noted, samples may be filtered to form each output pixel value. Output pixel values may be provided to projection devices PD_1 through PD_L for display on projection screen SCR. Output pixel values may also be provided to display device **84**. Sample buffer **162** may be configured to support super-sampling, critical sampling, or sub-sampling with respect to pixel resolution. In other words, the average distance between samples (X_k,Y_k) in the virtual image (stored in sample buffer **162**) may be smaller than, equal to, or larger than the average distance between pixel centers in virtual screen space. Furthermore, because the convolution kernel f(X,Y) may take non-zero functional values over a neighborhood which spans several pixel centers, a single sample may contribute to several output pixel values.

Sample memories **160A**–**160N** may comprise any of various types of memories (e.g., SDRAMs, SRAMs, RDRAMs, 3DRAMs, or next-generation 3DRAMs) in varying sizes. In one embodiment, each schedule unit **154** is coupled to four banks of sample memories, wherein each bank comprises four 3DRAM-64 memories. Together, the 3DRAM-64 memories may form a 116-bit deep super-sampled sample buffer that stores multiple samples per pixel. For example, in one embodiment, each sample memory **160A**–**160N** may store up to sixteen samples per pixel.

3DRAM-64 memories are specialized memories configured to support full internal double buffering with single buffered Z in one chip. The double buffered portion comprises two RGBX buffers, wherein X is a fourth channel that can be used to store other information (e.g., alpha). 3DRAM-64 memories also have a lookup table that takes in window ID information and controls an internal **2-1** or **3-1** multiplexer that selects which buffer's contents will be output. 3DRAM-64 memories are next-generation 3DRAM memories that may soon be available from Mitsubishi Electric Corporation's Semiconductor Group. In one embodiment, four chips used in combination are sufficient to create a double-buffered 1280x1024 super-sampled sample buffer.

Since the 3DRAM-64 memories are internally double-buffered, the input pins for each of the two frame buffers in the double-buffered system are time multiplexed (using multiplexers within the memories). The output pins may similarly be time multiplexed. This allows reduced pin count while still providing the benefits of double buffering. 3DRAM-64 memories further reduce pin count by not having z output pins. Since z comparison and memory buffer selection are dealt with internally, use of the 3DRAM-64 memories may simplify the configuration of sample buffer **162**. For example, sample buffer **162** may require little or no selection logic on the output side of the 3DRAM-64 memories. The 3DRAM-64 memories also reduce memory bandwidth since information may be written into a 3DRAM-64 memory without the traditional process of reading data out, performing a z comparison, and then writing data back in. Instead, the data may be simply written into the 3DRAM-64 memory, with the memory performing the steps described above internally.

However, in other embodiments of graphics system **112**, other memories (e.g., SDRAMs, SRAMs, RDRAMs, or current generation 3DRAMs) may be used to form sample buffer **162**.

Graphics processing unit **90** may be configured to generate a plurality of sample positions according to a particular sample positioning scheme (e.g., a regular grid, a perturbed regular grid, etc.). Alternatively, the sample positions (or offsets that are added to regular grid positions to form the sample positions) may be read from a sample position memory (e.g., a RAM/ROM table). Upon receiving a polygon that is to be rendered, graphics processing unit **90** determines which samples fall within the polygon based upon the sample positions. Graphics processing unit **90** renders the samples that fall within the polygon and stores rendered samples in sample memories **160**A–N. Note as used herein the terms render and draw are used interchangeably and refer to calculating color values for samples. Depth values, alpha values, and other per-sample values may also be calculated in the rendering or drawing process.

F. Sample-to-pixel Calculation Units

Sample-to-pixel calculation units **170-1** through **170**-V (collectively referred to as sample-to-pixel calculation units **170**) may be coupled between sample memories **160**A–N and DACs **178-1** through **178**-L. Sample-to-pixel calculation units **170** are configured to read selected samples from sample memories **160**A–N and then perform a convolution (i.e. a filtering operation) on the samples to generate the output pixel values which are provided to DACs **178-1** through **178**-L. The sample-to-pixel calculation units **170** may be programmable to allow them to perform different filter functions at different times, depending upon the type of output desired. In one embodiment, the sample-to-pixel calculation units **170** may implement a 5x5 super-sample reconstruction band-pass filter to convert the super-sampled sample buffer data (stored in sample memories **160**A–N) to pixel values. In other embodiments, calculation units **170** may filter a selected number of samples to calculate an output pixel. The selected samples may be multiplied by a spatial weighting function that gives weights to samples based on their position with respect to the center of the pixel being calculated. The filtering operation may use any of a variety of filters, either alone or in combination. For example, the convolution operation may employ a tent filter, a circular filter, an elliptic filter, a Mitchell filter, a band pass filter, a sync function filter, etc.

Sample-to-pixel calculation units **170** may also be configured with one or more of the following features: color look-up using pseudo color tables, direct color, inverse gamma correction, filtering of samples to pixels, programmable gamma encoding, and optionally color space conversion. Other features of sample-to-pixel calculation units **170** may include programmable video timing generators, programmable pixel clock synthesizers, edge-blending functions, hotspot correction functions, color space and crossbar functions. Once the sample-to-pixel calculation units have manipulated the timing and color of each pixel, the pixels are output to DACs **178-1** through **178**-L.

G. DACs

Digital-to-Analog Converters (DACs) **178-1** through **178**-L (collectively referred to as DACs **178**) operate as the final output stage of graphics system **112**. DACs **178** translate digital pixel data received from calculation units **170** into analog video signals. Each of DACs **178-1** through **178**-L may be coupled to a corresponding one of projections devices PD_I through PD_L. DAC **178-1** receives a first stream of digital pixel data from one or more of calculation units **170**, and converts the first stream into a first video signal. The first video signal is provided to projection device PD_I. Similarly, each of DACs **178-1** through **178**-L receive a corresponding stream of digital pixel data, and convert the digital pixel data stream into a corresponding analog video signal which is provided to a corresponding one of projection devices PD_I through PD_L.

Note in one embodiment DACs **178** may be bypassed or omitted completely in order to output digital pixel data in lieu of analog video signals. This may be useful projection devices PD_I through PD_L are based on a digital technology (e.g., an LCD-type display or a digital micro-mirror display).

Super-Sampling—FIGS. 4–5

FIG. **4** illustrates a portion of virtual screen space in a non-super-sampled example. The dots denote sample locations, and the rectangular boxes superimposed on virtual screen space define pixel boundaries. One sample is located in the center of each pixel, and values of red, green, blue, z, etc. are computed for the sample. For example, sample **74** is assigned to the center of pixel **70**. Although rendering units **150** may compute values for only one sample per pixel, sample-to-pixel calculation units **170** may still compute output pixel values based on multiple samples, e.g. by using a convolution filter whose support spans several pixels.

Turning now to FIG. **5**A, an example of one embodiment of super-sampling is illustrated. In this embodiment, two samples are computed per pixel. The samples are distributed according to a regular grid. Even through there are more samples than pixels in the figure, output pixel values could be computed using one sample per pixel, e.g. by throwing out all but the sample nearest to the center of each pixel. However, a number of advantages arise from computing pixel values based on multiple samples.

A support region **72** is superimposed over pixel **70**, and illustrates the support of a filter which is localized at pixel **70**. The support of a filter is the set of locations over which the filter (i.e. the filter kernel) takes non-zero values. In this example, the support region **72** is a circular disc. The output pixel values (e.g. red, green, blue and z values) for pixel **70** are determined only by samples **74**A and **74**B, because these are the only samples which fall within support region **72**.

This filtering operation may advantageously improve the realism of a displayed image by smoothing abrupt edges in the displayed image (i.e., by performing anti-aliasing). The filtering operation may simply average the values of samples **74**A–B to form the corresponding output values of pixel **70**, or it may increase the contribution of sample **74**B (at the center of pixel **70**) and diminish the contribution of sample **74**A (i.e., the sample farther away from the center of pixel **70**). The filter, and thus support region **72**, is repositioned for each output pixel being calculated so the center of support region **72** coincides with the center position of the pixel being calculated. Other filters and filter positioning schemes are also possible and contemplated.

In the example of FIG. **5**A, there are two samples per pixel. In general, however, there is no requirement that the number of samples be related to the number of pixels. The number of samples may be completely independent of the number of pixels. For example, the number of samples may be smaller than the number of pixels. (This is the condition that defines sub-sampling).

Turning now to FIG. **5**B, another embodiment of super-sampling is illustrated. In this embodiment, the samples are positioned randomly. Thus, the number of samples used to calculate output pixel values may vary from pixel to pixel. Render units **150**A–D calculate color information at each sample position.

Super-Sampled Sample Buffer with Real-Time Sample-To-Pixel Calculation—FIGS. 6–10

FIG. **6** illustrates one possible configuration for the flow of data through one embodiment of graphics system **112**. As the figure shows, geometry data **350** is received by graphics system **112** and used to perform draw process **352**. The draw process **352** is implemented by one or more of control unit **140**, rendering units **150**, data memories **152**, and schedule unit **154**. Geometry data **350** comprises data for one or more polygons. Each polygon comprises a plurality of vertices (e.g., three vertices in the case of a triangle), some of which may be shared among multiple polygons. Data such as x, y, and z coordinates, color data, lighting data and texture map information may be included for each vertex.

In addition to the vertex data, draw process **352** (which may be performed by rendering units **150**A–D) also receives sample position information from a sample position memory **354**. The sample position information defines the location of samples in virtual screen space, i.e. in the 2-D viewport. Draw process **352** selects the samples that fall within the polygon currently being rendered, calculates a set of values (e.g. red, green, blue, z, alpha, and/or depth of field information) for each of these samples based on their respective positions within the polygon For example, the z value of a sample that falls within a triangle may be interpolated from the known z values of the three vertices. Each set of computed sample values are stored into sample buffer **162**.

In one embodiment, sample position memory **354** is embodied within rendering units **150**A–D. In another embodiment, sample position memory **354** may be realized as part of memories **152**A–**152**D, or as a separate memory.

Sample position memory **354** may store sample positions in terms of their virtual screen coordinates (X,Y). Alternatively, sample position memory **354** may be configured to store only offsets dX and dY for the samples with respect to positions on a regular grid. Storing only the offsets may use less storage space than storing the entire coordinates (X,Y) for each sample. The sample position information stored in sample position memory **354** may be read by a dedicated sample position calculation unit (not shown) and processed to calculate sample positions for graphics processing unit **90**. More detailed information on the computation of sample positions is included below (see description of FIGS. **9** and **10**).

In another embodiment, sample position memory **354** may be configured to store a table of random numbers. Sample position memory **354** may also comprise dedicated hardware to generate one or more different types of regular grids. This hardware may be programmable. The stored random numbers may be added as offsets to the regular grid positions generated by the hardware. In one embodiment, sample position memory **354** may be programmable to access or "unfold" the random number table in a number of different ways, and thus, may deliver more apparent randomness for a given length of the random number table. Thus, a smaller table may be used without generating the visual artifacts caused by simple repetition of sample position offsets.

Sample-to-pixel calculation process **360** uses the same sample positions as draw process **352**. Thus , in one embodiment, sample position memory **354** may generate a sequence of random offsets to compute sample positions for draw process **352**, and may subsequently regenerate the same sequence of random offsets to compute the same sample positions for sample-to-pixel calculation process **360**. In other words, the unfolding of the random number table may be repeatable. Thus, it may not be necessary to store sample positions at the time of their generation for draw process **352**.

As shown in FIG. **6**, sample position memory **354** may be configured to store sample offsets generated according to a number of different schemes such as a regular square grid, a regular hexagonal grid, a perturbed regular grid, or a random (stochastic) distribution. Graphics system **112** may receive an indication from the operating system, device driver, or the geometry data **350** that indicates which type of sample positioning scheme is to be used. Thus the sample position memory **354** is configurable or programmable to generate position information according to one or more different schemes. More detailed information on several sample positioning schemes are described further below (see description of FIG. **8**).

In one embodiment, sample position memory **354** may comprise a RAM/ROM that contains stochastically determined sample points or sample offsets. Thus, the density of samples in virtual screen space may not be uniform when observed at small scale. Two bins with equal area centered at different locations in virtual screen space may contain different numbers of samples. As used herein, the term "bin" refers to a region or area in virtual screen space.

An array of bins may be superimposed over virtual screen space, i.e. the 2-D viewport, and the storage of samples in sample buffer **162** may be organized in terms of bins. The sample buffer **162** may comprise an array of memory blocks which correspond to the bins. Each memory block may store the sample values (e.g. red, green, blue, z, alpha, etc.) for the samples that fall within the corresponding bin. The approximate location of a sample is given by the bin in which it resides. The memory blocks may have addresses which are easily computable from the corresponding bin locations in virtual screen space, and vice versa. Thus, the use of bins may simplify the storage and access of sample values in sample buffer **162**.

The bins may tile the 2-D viewport in a regular array, e.g. in a square array, rectangular array, triangular array, hexagonal array, etc., or in an irregular array. Bins may occur in a variety of sizes and shapes. The sizes and shapes may be programmable. The maximum number of samples that may populate a bin is determined by the storage space allocated to the corresponding memory block. This maximum number of samples is referred to herein as the bin sample capacity, or simply, the bin capacity. The bin capacity may take any of a variety of values. The bin capacity value may be programmable. Henceforth, the memory blocks in sample buffer **162** which correspond to the bins in virtual screen space will be referred to as memory bins.

The specific position of each sample within a bin may be determined by looking up the sample's offset in the RAM/ROM table, i.e. the sample's offset with respect to the bin position (e.g. the lower-left corner or center of the bin, etc.). However, depending upon the implementation, not all choices for the bin capacity may have a unique set of offsets stored in the RAM/ROM table. Offsets for a first bin capacity value may be determined by accessing a subset of the offsets stored for a second larger bin capacity value. In one embodiment, each bin capacity value supports at least four different sample positioning schemes. The use of different sample positioning schemes may reduce final image artifacts due to repeating sample positions.

In one embodiment, sample position memory **354** may store pairs of **8**-bit numbers, each pair comprising an x-offset and a y-offset. (Other offsets are also possible, e.g., a time offset, a z-offset, etc.) When added to a bin position, each pair defines a particular position in virtual screen space, i.e. the 2-D viewport. To improve read access times, sample position memory **354** may be constructed in a wide/parallel manner so as to allow the memory to output more than one sample location per read cycle.

Once the sample positions have been read from sample position memory **354**, draw process **352** selects the samples that fall within the polygon currently being rendered. Draw process **352** then calculates the z and color information (which may include alpha or other depth of field information values) for each of these samples and stores the data into sample buffer **162**. In one embodiment, sample buffer **162** may only single-buffer z values (and perhaps alpha values) while double-buffering other sample components such as color. Unlike prior art systems, graphics system **112** may use double-buffering for all samples (although not all components of samples may be double-buffered, i.e., the samples may have some components that are not double-buffered). In one embodiment, the samples are stored into sample buffer **162** in bins. In some embodiments, the bin capacity may vary from frame to frame. In addition, the bin capacity may vary spatially for bins within a single frame rendered into sample buffer **162**. For example, bins on the edge of the 2-D viewport may have a smaller bin capacity than bins corresponding to the center of the 2-D viewport. Since viewers are likely to focus their attention mostly on the center of the screen SCR or display image DIM, more processing bandwidth may be dedicated to providing enhanced image quality in the center of 2-D viewport. Note that the size and shape of bins may also vary from region to region, or from frame to frame. The use of bins will be described in greater detail below.

In parallel and independently of draw process **352**, filter process **360** is configured to: (a) read sample positions from sample position memory **354**, (b) read corresponding sample values from sample buffer **162**, (c) filter the sample values, and (d) output the resulting output pixel values to one or more of projection devices PD_l through PD_L and/or display device **84**. Sample-to-pixel calculation units **170** implement filter process **360**. Filter process **360** is operable to generate the red, green, and blue values for an output pixel based a spatial filtering of the corresponding data for a selected plurality of samples, e.g. samples falling in a neighborhood of the pixel center. Other values such as alpha may also be generated. In one embodiment, filter process **360** is configured to: (i) determine the distance of each sample from the pixel center; (ii) multiply each sample's attribute values (e.g., red, green, blue, alpha) by a filter weight that is a specific (programmable) function of the sample's distance; (iii) generate sums of the weighted attribute values, one sum per attribute (e.g. a sum for red, a sum for green, etc.), and (iv) normalize the sums to generate the corresponding pixel attribute values. Filter process **360** is described in greater detail below (see description accompanying FIGS. **11**, **12**, and **14**).

In the embodiment just described, the filter kernel is a function of distance from the pixel center, and thus, is radially symmetric. However, in alternative embodiments, the filter kernel may be a more general function of X and Y displacements from the pixel center. Thus, the support of the filter, i.e. the 2-D neighborhood over which the filter kernel takes non-zero values, may not be a circular disk. Any sample falling within the support of the filter kernel may affect the output pixel being computed.

Turning now to FIG. **7**, a diagram illustrating an alternate embodiment of graphics system **112** is shown. In this embodiment, two or more sample position memories **354A** and **354B** are utilized. Thus, the sample position memories **354A–B** are essentially double-buffered. If the sample positions remain the same from frame to frame, then the sample positions may be single-buffered. However, if the sample positions vary from frame to frame, then graphics system **112** may be advantageously configured to double-buffer the sample positions. The sample positions may be double-buffered on the rendering side (i.e., memory **354A** may be double-buffered) and/or the filter side (i.e., memory **354B** may be double-buffered). Other combinations are also possible. For example, memory **354A** may be single-buffered, while memory **354B** is doubled-buffered. This configuration may allow one side of memory **354B** to be updated by draw process **352** while the other side of memory **354B** is accessed by filter process **360**. In this configuration, graphics system **112** may change sample positioning schemes on a per-frame basis by shifting the sample positions (or offsets) from memory **354A** to double-buffered memory **354B** as each frame is rendered. Thus, the sample positions which are stored in memory **354A** and used by draw process **352** to render sample values may be copied to memory **354B** for use by filter process **360**. Once the sample position information has been copied to memory **354B**, position memory **354A** may then be loaded with new sample positions (or offsets) to be used for a second frame to be rendered. In this way the sample position information follows the sample values from the draw **352** process to the filter process **360**.

Yet another alternative embodiment may store tags to offsets with the sample values in super-sampled sample buffer **162**. These tags may be used to look-up the offset (i.e. perturbations) dX and dY associated with each particular sample.

Sample Positioning Schemes

FIG. **8** illustrates a number of different sample positioning schemes. In the regular positioning scheme **190**, samples are positioned at fixed positions with respect to a regular grid which is superimposed on the 2-D viewport. For example, samples may be positioned at the center of the rectangles which are generated by the regular grid. More generally, any tiling of the 2-D viewport may generate a regular positioning scheme. For example, the 2-D viewport may be tiled with triangles, and thus, samples may be positioned at the centers (or vertices) of the triangular tiles. Hexagonal tilings, logarithmic tilings, and semi-regular tilings such as Penrose tilings are also contemplated.

In the perturbed regular positioning scheme **192**, sample positions are defined in terms of perturbations from a set of fixed positions on a regular grid or tiling. In one embodiment, the samples may be displaced from their corresponding fixed grid positions by random x and y offsets, or by random angles (ranging from 0 to 360 degrees) and random radii (ranging from zero to a maximum radius). The offsets may be generated in a number of ways, e.g. by hardware based upon a small number of seeds, by reading a table of stored offsets, or by using a pseudo-random function. Once again, perturbed regular gird scheme **192** may be based on any type of regular grid or tiling. Samples generated by perturbation with respect to a grid (e.g., hexagonal tiling may particularly desirable due to the geometric properties of this configuration).

Stochastic sample positioning scheme **194** represents a third potential type of scheme for positioning samples. Stochastic sample positioning involves randomly distributing the samples across the 2-D viewport. Random positioning of samples may be accomplished through a number of different methods, e.g., using a random number generator such as an internal clock to generate pseudo-random numbers. Random numbers or positions may also be precalculated and stored in memory. Note, as used in this application, random positions may be selected from a statistical population (e.g., a Poisson-disk distribution). Different types of random and pseudo-random positions are described in greater detail in Chapter 10 of Volume 1 of the treatise titled "Principles of Digital Image Synthesis" by Andrew S. Glassner, Morgan Kaufman Publishers 1995.

Turning now to FIG. **9**, details of one embodiment of perturbed regular positioning scheme **192** are shown. In this embodiment, samples are randomly offset from a regular square grid by x- and y-offsets. As the enlarged area shows, sample **198** has an x-offset **134** that specifies its horizontal displacement from its corresponding grid intersection point **196**. Similarly, sample **198** also has a y-offset **136** that specifies its vertical displacement from grid intersection point **196**. The random x-offset **134** and y-offset **136** may be limited to a particular range of values. For example, the x-offset may be limited to the range from zero to X_{max} where X_{max} is the width of the a grid rectangle. Similarly, the y-offset may be limited to the range from zero to Y_{max} may Y_{max} is the height of a grid rectangle. The random offset may also be specified by an angle and radius with respect to the grid intersection point **196**.

FIG. **10** illustrates details of another embodiment of the perturbed regular grid scheme **192**. In this embodiment, the samples are grouped into rectangular bins **138A–D**. In this embodiment, each bin comprises nine samples, i.e. has a bin capacity of nine. Different bin capacities may be used in other embodiments (e.g., bins storing four samples, 16 samples, etc.). Each sample's position may be determined by an x- and y-offset relative to the origin of the bin in which it resides. The origin of a bin may be chosen to be the lower-left corner of the bin (or any other convenient location within the bin). For example, the position of sample **198** is determined by summing x-offset **124** and y-offset **126** respectively to the x and y coordinates of the origin **132D** of bin **138D**. As previously noted, this may reduce the size of sample position memory **354** used in some embodiments.

FIG. 11—Converting Samples into Pixels

As discussed earlier, the 2-D viewport may be covered with an array of spatial bins. Each spatial bin may be populated with samples whose positions are determined by sample position memory **354**. Each spatial bin corresponds to a memory bin in sample buffer **162**. A memory bin stores the sample values (e.g. red, green, blue, z, alpha, etc.) for the samples that reside in the corresponding spatial bin. Sample-to-pixel calculation units **170** (also referred to as convolve units **170**) are configured to read memory bins from sample buffer **162** and to convert sample values contained within the memory bins into pixel values.

Parallel Sample-to-Pixel Filtering using Columns—FIGS. 11A–11B

FIG. **11**A illustrates one method for rapidly converting sample values stored in sample buffer **162** into pixel values. The spatial bins which cover the 2-D viewport may be organized into columns (e.g., Cols. **1**–**4**). Each column comprises a two-dimensional sub-array of spatial bins. The columns may be configured to horizontally overlap (e.g., by one or more bins). Each of the sample-to-pixel calculation units **170-1** through **170-4** may be configured to access memory bins corresponding to one of the columns. For example, sample-to-pixel calculation unit **170-1** may be configured to access memory bins that correspond to the spatial bins of Column **1**. The data pathways between sample buffer **162** and sample-to-pixel calculations unit **170** may be optimized to support this column-wise correspondence.

The amount of the overlap between columns may depend upon the horizontal diameter of the filter support for the filter kernel being used. The example shown in FIG. **11**A illustrates an overlap of two bins. Each square (such as square **188**) represents a single bin comprising one or more samples. Advantageously, this configuration may allow sample-to-pixel calculation units **170** to work independently and in parallel, with each of the sample-to-pixel calculation units **170** receiving and convolving samples residing in the memory bins of the corresponding column. Overlapping the columns will prevent visual bands or other artifacts from appearing at the column boundaries for any operators larger than a pixel in extent.

Furthermore, the embodiment of FIG. **11**A includes a plurality of bin caches **176** which couple to sample buffer **162**. In addition, each of bin caches **176** couples to a corresponding one of sample-to-pixel calculation units **170**. Generic bin cache **176**-I (where I takes any value positive integer value) stores a collection of memory bins corresponding to Column I and serves as a cache for sample-to-pixel calculation unit **170**-I. Generic bin cache **176**-I may have an optimized coupling to sample buffer **162** which facilitates access to the memory bins for Column I. Since the sample-to-pixel calculation for two adjacent output pixels may involve many of the same bins, bin caches **176** may increase the overall access bandwidth to sample buffer **162**. Sample-to-bin calculation units **170** may be implemented in a number of different ways, including using high performance ALU (arithmetic logic unit) cores, functional units from a microprocessor or DSP, or a custom design that uses hardware multipliers and adders.

Turning now to FIG. **11**B, another method for performing parallel sample-to-pixel calculation is shown. In this embodiment, sample buffer **162** is divided into a plurality of vertical columns or stripes as in the previously described embodiment. However, the columns in this embodiment are not of equal size or width. For example, column one may contain significantly fewer bins of samples than column four. This embodiment may be particularly useful in configurations of the graphics system that support variable sample densities. As previously noted and as described in greater detail below, the graphics system may devote more samples (i.e., a higher sample density) for areas of sample buffer **162** that correspond to areas of the final image that would benefit the most from higher sample densities, e.g., areas of particular interest to the viewer or areas of the image that correspond to the viewer's point of foveation (described in greater detail below). In these systems that support variable sample densities, the ability to vary the widths of the columns may advantageously allow the graphics system to equalize the number of samples filtered by each of the sample-to-pixel calculation units. For example, column one may correspond to a portion of the displayed image upon which the center of the viewers view point is focused. Thus, the graphics system may devote a high density of samples to the bins in column one, and the graphics system may devote a lower density of samples to the bins in column four. Thus, by decreasing the width of column one and increasing the width of column four, sample-to-pixel calculation units **170-1** and **170-4** may each filter approximately the same number of samples. Advantageously, balancing the filtering load among the sample-to-pixel calculation units may allow the graphics system to use the processing resources of the sample-to-pixel calculation units in a more efficient manner.

In some embodiments, the graphics system may be configured to dynamically change the widths of the columns on a frame by frame basis (or even on a fraction of a frame basis). In embodiments of the graphics system that change sample densities dynamically (e.g., eye-tracking, point of foveation tracking, main character tracking), the sample densities may vary on a frame by frame basis, thus varying the column width on a frame by frame basis once again allows the computing resources of sample-to-pixel calculation units **170** to be utilized in a more efficient manner. In some embodiments, the column width may be varied on a scan line basis or some other time-basis. In addition to varying with time, as the figure illustrates the columns may also be configured to overlap (as in the previously described embodiment) to prevent the appearance of any visual artifacts (e.g., seams, tears, or vertical lines).

Parallel Sample-to-Pixel Filtering using Rows—FIG. 12

Turning now to FIG. **12**, another embodiment of the graphics system is shown. In this embodiment, sample buffer **162** is divided into a plurality of horizontal rows or stripes. As with the previous embodiments, the rows may overlap and/or vary in width to compensate for varying sample densities. As with the previous embodiment, each row may provide bins (and samples) to a particular bin cache **176** and corresponding sample-to-pixel calculation unit **170**.

Parallel Sample-to-Pixel Filtering using Rows—FIG. **13**

Turning now to FIG. **13**, yet another embodiment of the graphics system is shown. In this embodiment, sample buffer **162** is divided into a plurality of rectangular regions. As with the previous embodiments, the rectangular regions may or may not overlap, have different sizes, and/or dynamically vary in size (e.g., on a frame by frame or scan line basis). Each region may be configured to provide bins (and samples) to a particular bin cache **176** and corresponding sample-to-pixel calculation unit **170**. In some embodiments, each rectangular region may correspond to the image projected by one of a plurality of projectors (e.g., LCD projectors). In other embodiments, each rectangular region may correspond to a particular portion of a single image projected or displayed on a single display device. As with the previous embodiments, advantageously, the sample-to-pixel calculation units **170** may be configured to operate independently and in parallel, thereby reducing the graphics systems' latency. As previously noted, the rectangular regions illustrated in FIG. **13** need not be of uniform size and/or shape.

In embodiments of the graphics system that have varying region sizes or stripe widths, the amount of overlap may also vary dynamically on a frame by frame or sub-frame basis. Note, other shapes for the regions into which sample buffer **162** may be divided are possible and contemplated. For example, in some embodiments each sample-to-pixel calculation unit may receive bins (and samples) from multiple small regions or stripes.

In some embodiments, sample caches **176** may not have enough storage space to store an entire horizontal scan line. For this reason dividing the sample buffer into regions may be useful. Depending on the display device, the regions may be portions of odd only and even only scan lines. In some systems, e.g. those with multiple display devices, each region may correspond to a single display device or to a quadrant of an image being displayed. For example, assuming the images formed by four projectors are tiled together to form a single, large image, then each sample-to-pixel calculation unit could receive samples corresponding to pixels displayed by a particular projector. In some embodiments, the overlapping areas of the regions may be stored twice, thereby allowing each sample-to-pixel calculation unit exclusive access to a particular region of the sample buffer. This may prevent timing problems that result when two different sample-to-pixel calculation units (or two sample cache controllers) attempt to access the same set of memory locations at the same time. In other embodiments the sample buffer may be multi-ported to allow one or more multiple concurrent accesses to the same memory locations.

As previously noted, in some embodiments the sample caches are configured to read samples from the sample buffer. In some embodiments, the samples may be read on a bin-by-bin basis from the sample buffer. The sample cache and/or sample buffer may include control logic that is configured to ensure that all samples that have a potential to contribute to one or more pixels that are being filtered (or that are about to be filtered) are available for the corresponding sample-to-pixel calculation unit. In some implementations, the sample caches may be large enough to store a predetermined array of bins such as 5×5 bins (e.g., to match the maximum filter size). In another embodiment, instead of a 5×5 bin cache, the sample caches may be configured to output pixels as they are being accumulated to a series of multiple accumulators. In this embodiment, a different coefficient is generated for each pixel, depending upon the number of samples and their weightings.

Method for Reading Samples from Sample Buffer—FIG. 14

Turning now to FIG. **14**, more details of one embodiment of a method for reading sample values from a super-sampled sample buffer are shown. As the figure illustrates, the sample-to-pixel filter kernel **400** travels across Column I (in the direction of arrow **406**) to generate output pixel values, where index I takes any value in the range from one to four. Sample-to-pixel calculation unit **170**-I may implement the sample-to-pixel filter kernel **400**. Bin cache **176**-I may used to provide fast access to the memory bins corresponding to Column I. For example, bin cache **176**-I may have a capacity greater than or equal to 25 memory bins since the support of sample-to-pixel filter kernel **400** covers a 5 by 5 array of spatial bins. As the sample-to-pixel operation proceeds, memory bins are read from the super-sampled sample buffer **162** and stored in bin cache **176**-I. In one embodiment, bins that are no longer needed, e.g. bins **410**, are overwritten in bin cache **176**-I by new bins. As each output pixel is generated, sample-to-pixel filter kernel **400** shifts. Kernel **400** may be visualized as proceeding in a sequential fashion within Column I in the direction indicated by arrow **406**. When kernel **400** reaches the right boundary **404** of the Column I, it may shift down one or more rows of bins, and then, proceed horizontally starting from the left column boundary **402**. Thus the sample-to-pixel operation proceeds in a scan line manner generating successive rows of output pixels for display.

FIG. **15** illustrates potential border conditions in the computation of output pixel values. The 2-D viewport **420** is illustrated as a rectangle which is overlaid with a rectangular array of spatial bins. Recall that every spatial bin corresponds to a memory bin in sample buffer **162**. The memory bin stores the sample values and/or sample positions for samples residing in the corresponding spatial bin. As described above, sample-to-pixel calculation units **170** filter samples in the neighborhood of a pixel center in order to generate output pixel values (e.g. red, green, blue, etc.). Pixel center PC_0 is close enough to the lower boundary (Y=0) of the 2-D viewport **420** that its filter support **400** is not entirely contained in the 2-D viewport. Sample-to-pixel calculation units **170** may generate sample positions and/or sample values for the marginal portion of filter support **400** (i.e. the portion which falls outside the 2-D viewport **420**) according to a variety of methods.

In one embodiment, sample-to-pixel calculation units **170** may generate one or more dummy bins to cover the marginal area of the filter support **400**. Sample positions for the dummy bins may be generated by reflecting the sample positions of spatial bins across the 2-D viewport boundary. For example, dummy bins F, G, H, I and J may be assigned sample positions by reflecting the sample positions corresponding to spatial bins A, B, C, D, and E respectively, across the boundary line Y=0. Predetermined color values may be associated with these dummy samples in the dummy bins. For example, the value (0,0,0) for the RGB color vector may be assigned to each dummy sample. As pixel center PC_0 moves downward (i.e. toward the boundary Y=0 and through it), additional dummy bins with dummy samples may be generated to cover filter support **400** (which moves along with the pixel center PC_0. The number of dummy samples falling within filter support **400** increases and reaches a maximum when filter support **400** has moved entirely outside of the 2-D viewport **420**. Thus, the color value computed based on filter support **400** approaches the predetermined background color as the pixel center PC_0 crosses the boundary.

A pixel center may lie outside of the 2-D viewport **420**, and yet, may be close enough to the viewport boundary so that part of its filter support lies in the 2-D viewport **420**. Filter support **401** corresponds to one such pixel center. Sample-to-pixel calculation units **170** may generate dummy bins Q, R, S, T, U and V to cover the external portion of filter support **401** (i.e. the portion external to the 2-D viewport). The dummy bins Q, R and S may be assigned sample positions based on the sample positions of spatial bins N, O and P, and/or spatial bins K, L and M.

The sample positions for dummy bins may also be generated by translating the sample positions corresponding to spatial bins across the viewport boundary, or perhaps, by generating sample positions on-the-fly according to a regular, a perturbed regular or stochastic sample positioning scheme.

FIG. **16** illustrates an alternate embodiment of a method for performing pixel value computations. Sample-to-pixel computation units **170** may perform pixel value computations using a viewable subwindow **422** of the 2-D viewport **420**. The viewable subwindow is depicted as a rectangle with lower left corner at (X_1,Y_1) and upper right corner at (X_2,Y_2) in virtual screen space. Note, in some embodiments the filter may be auto-normalized or pre-normalized to reduce the number of calculations required for determining the final pixel value.

Rendering Samples into a Super-Sampled Sample Buffer—FIG. 17

FIG. **17** is a flowchart of one embodiment of a method for drawing or rendering samples into a super-sampled sample buffer. Certain of the steps of FIG. **17** may occur concurrently or in different orders. In step **200**, graphics system **112** receives graphics commands and graphics data from the host CPU **102** or directly from system memory **106**. In step **202**, the instructions and data are routed to one or more of rendering units **150**A–D. In step **204**, rendering units **150**A–D determine if the graphics data is compressed. If the graphics data is compressed, rendering units **150**A–D decompress the graphics data into a useable format, e.g., triangles, as shown in step **206**. Next, the triangles are processed, e.g., converted from model space to world space, lit, and transformed (step **208**A).

If the graphics system implements variable resolution super-sampling, then the triangles are compared with a set of sample-density region boundaries (step **208**B). In variable-resolution super-sampling, different regions of the 2-D viewport may be allocated different sample densities based upon a number of factors (e.g., the center of the attention of an observer on projection screen SCR as determined by eye or head tracking). Sample density regions are described in greater detail below (see section entitled Variable Resolution Sample Buffer below). If the triangle crosses a sample-density region boundary (step **210**), then the triangle may be divided into two smaller polygons along the region boundary (step **212**). The polygons may be further subdivided into triangles if necessary (since the generic slicing of a triangle gives a triangle and a quadrilateral). Thus, each newly formed triangle may be assigned a single sample density. In one embodiment, graphics system **112** may be configured to render the original triangle twice, i.e. once with each sample density, and then, to clip the two versions to fit into the two respective sample density regions.

In step **214**, one of the sample positioning schemes (e.g., regular, perturbed regular, or stochastic) is selected from sample position memory **354**. The sample positioning scheme will generally have been pre-programmed into the sample position memory **354**, but may also be selected "on the fly". In step **216**, rendering units **150**A–D determine which spatial bins may contain samples located within the triangle's boundaries, based upon the selected sample positioning scheme and the size and shape of the spatial bins. In step **218**, the offsets dX and dY for the samples within these spatial bins are then read from sample position memory **354**. In step **220**, each sample's position is then calculated using the offsets dX and dY and the coordinates of the corresponding bin origin, and is compared with the triangle's vertices to determine if the sample is within the triangle. Step **220** is discussed in greater detail below.

For each sample that is determined to be within the triangle, the rendering unit draws the sample by calculating the sample's color, alpha and other attributes. This may involve a lighting calculation and an interpolation based upon the color and texture map information associated with the vertices of the triangle. Once the sample is rendered, it may be forwarded to schedule unit **154**, which then stores the sample in sample buffer **162** (step **224**).

Note the embodiment of the rendering method described above is used for explanatory purposes only and is not meant to be limiting. For example, in some embodiments, the steps shown in FIG. **13** as occurring serially may be implemented in parallel. Furthermore, some steps may be reduced or eliminated in certain embodiments of the graphics system (e.g., steps **204**–**206** in embodiments that do not implement geometry compression, or steps **210**–**212** in embodiments that do not implement a variable resolution super-sampled sample buffer).

Determination of Which Samples are in Polygon Being Rendered—FIG. 18

The determination of which samples reside within the polygon being rendered may be performed in a number of different ways. In one embodiment, the deltas between the three vertices defining the triangle are first determined. For example, these deltas may be taken in the order of first to second vertex (v2 v1)=d**12**, second to third vertex (v**3**–v**2**)=d**23**, and third vertex back to the first vertex (v**1**–v**3**)=d**31**. These deltas form vectors, and each vector may be categorized as belonging to one of the four quadrants of the coordinate plane (e.g., by using the two sign bits of its delta X and Y components). A third condition may be added determining whether the vector is an X-major vector or Y-major vector. This may be determined by calculating whether abs(delta_x) is greater than abs(delta_y). Using these three bits of information, the vectors may each be categorized as belonging to one of eight different regions of the coordinate plane. If three bits are used to define these regions, then the X-sign bit (shifted left by two), the Y-sign bit (shifted left by one), and the X-major bit, may be used to create the eight regions as shown in FIG. **18**.

Next, three edge inequalities may be used to define the interior of the triangle. The edges themselves may be described as lines in the either (or both) of the forms y=mx+b or x=ry+c, where rm=1. To reduce the numerical range needed to express the slope, either the X-major and Y-major equation form for an edge equation may be used (so that the absolute value of the slope may be in the range of 0 to 1). Thus, the edge (or half-plane) inequalities may be expressed in either of two corresponding forms:

X-major: $y-m\cdot x-b<0$, when point (x,y) is below the edge;

Y-major: $x-r\cdot y-c<0$, when point (x,y) is to the left of the edge.

The X-major inequality produces a logical true value (i.e. sign bit equal to one) when the point in question (x,y) is below the line defined by the an edge. The Y-major equation produces a logical true value when the point in question (x,y) is to the left of the line defined by an edge. The side which comprises the interior of the triangle is known for each of the linear inequalities, and may be specified by a Boolean variable referred to herein as the accept bit. Thus, a sample (x,y) is on the interior side of an edge if

X-major: $(y-m\cdot x-b<0$<xor>accept=true;

Y-major: $(y-m\cdot x-b<0$<xor>accept=true;

The accept bit for a given edge may be calculated according to the following table based on (a) the region (zero through seven) in which the edge delta vector resides, and (b) the sense of edge traversal, where clockwise traversal is indicated by cw=1 and counter-clockwise traversal is indicated by cw=0. The notation "!" denotes the logical complement.

1: accept=!cw

0: accept=cw

4: accept=cw

5: accept=cw

7: accept=cw

6: accept=!cw

2: accept=!cw

3: accept=!cw

Tie breaking rules for this representation may also be implemented (e.g., coordinate axes may be defined as belonging to the positive octant). Similarly, X-major may be defined as owning all points that tie on the slopes.

In an alternate embodiment, the accept side of an edge may be determined by applying the edge inequality to the third vertex of the triangle (i.e. the vertex that is not one of the two vertices forming the edge). This method may incur the additional cost of a multiply-add, which may be avoided by the technique described above.

To determine the "faced-ness" of a triangle (i.e., whether the triangle is clockwise or counter-clockwise), the delta-directions of two edges of the triangle may be checked and the slopes of the two edges may be compared. For example, assuming that edge**12** has a delta-direction of **1** and the second edge (edge**23**) has a delta-direction of **0**, **4**, or **5**, then the triangle is counter-clockwise. If, however, edge**23** has a delta-direction of **3**, **2**, or **6**, then the triangle is clockwise. If edge**23** has a delta-direction of **1** (i.e., the same as edge**12**), then comparing the slopes of the two edges breaks the tie (both are x-major). If edge**12** has a greater slope, then the triangle is clockwise. If edge**23** has a delta-direction of **7** (the exact opposite of edge**12**), then again the slopes are compared, but with opposite results in terms of whether the triangle is clockwise or counter-clockwise.

The same analysis can be exhaustively applied to all combinations of edge**12** and edge**23** delta-directions, in every case determining the proper faced-ness. If the slopes are the same in the tie case, then the triangle is degenerate (i.e., with no interior area). It can be explicitly tested for and culled, or, with proper numerical care, it could be let through as it will cause no samples to render. One special case arises when a triangle splits the view plane. However, this case may be detected earlier in the pipeline (e.g., when front plane and back plane clipping are performed).

Note in most cases only one side of a triangle is rendered. Thus, if the faced-ness of a triangle determined by the analysis above is the one to be rejected, then the triangle can be culled (i.e., subject to no further processing with no samples generated). Further note that this determination of faced-ness only uses one additional comparison (i.e., of the slope of edge**12** to that of edge**23**) beyond factors already computed. Many traditional approaches may utilize more complex computations (though at earlier stages of the set-up computation).

Generating Output Pixels Values from Sample Values—FIG. 19

FIG. **19** is a flowchart of one embodiment of a method for selecting and filtering samples stored in super-sampled sample buffer **162** to generate output pixel values. In step **250**, a stream of memory bins are read from the super-sampled sample buffer **162**. In step **252**, these memory bins may be stored in one or more of bin caches **176** to allow the sample-to-pixel calculation units **170** easy access to sample values during the sample-to-pixel operation. In step **254**, the memory bins are examined to determine which of the memory bins may contain samples that contribute to the output pixel value currently being generated. Each sample that is in a bin that may contribute to the output pixel is then individually examined to determine if the sample does indeed contribute (steps **256**–**258**). This determination may be based upon the distance from the sample to the center of the output pixel being generated.

In one embodiment, the sample-to-pixel calculation units **170** may be configured to calculate this distance (i.e., the extent or envelope of the filter at the sample's position) and then use it to index into a table storing filter weight values according to filter extent (step **260**). In another embodiment, however, the potentially expensive calculation for determining the distance from the center of the pixel to the sample (which typically involves a square root function) is avoided by using distance squared to index into the table of filter weights. Alternatively, a function of x and y may be used in lieu of one dependent upon a distance calculation. In one embodiment, this may be accomplished by utilizing a floating point format for the distance (e.g., four or five bits of mantissa and three bits of exponent), thereby allowing much of the accuracy to be maintained while compensating for the increased range in values. In one embodiment, the table may be implemented in ROM. However, RAM tables may also be used. Advantageously, RAM tables may, in some embodiments, allow the graphics system to vary the filter coefficients on a per-frame basis. For example, the filter coefficients may be varied to compensate for known shortcomings of the display or for the user's personal preferences. In some embodiments, the use of RAM tables may allow the user to select different filters (e.g., via a sharpness control on the display device or in a window system control panel). A number of different filters may be implemented to generate desired levels of sharpness based on different display types. For example, the control panel may have one setting optimized of LCD displays and another setting optimized for CRT displays. The graphics system can also vary the filter coefficients on a screen area basis within a frame, or on a per-output pixel basis. Another alternative embodiment may actually calculate the desired filter weights for each sample using specialized hardware (e.g., multipliers and adders). The filter weight for samples outside the limits of the sample-to-pixel filter may simply be multiplied by a filter weight of zero (step **262**), or they may be removed from the calculation entirely.

Once the filter weight for a sample has been determined, the sample may then be multiplied by its filter weight (step **264**). The weighted sample may then be summed with a running total to determine the final output pixel's color value (step **266**). The filter weight may also be added to a running total pixel filter weight (step **268**), which is used to normalize the filtered pixels. Normalization advantageously prevents the filtered pixels (e.g., pixels with more samples than other pixels) from appearing too bright or too dark by compensating for gain introduced by the sample-to-pixel calculation process. After all the contributing samples have been weighted and summed, the total pixel filter weight may be used to divide out the gain caused by the filtering (step **270**). Finally, the normalized output pixel may be output and/or processed through one or more of the following processes (not necessarily in this order): gamma correction, color look-up using pseudo color tables, direct color, inverse gamma correction, programmable gamma encoding, color space conversion, and digital-to-analog conversion, before eventually being displayed (step **274**).

In some embodiments, the graphics system may be configured to use each sample's alpha information to generate a mask that output with the sample. The mask may be used to perform real-time soft-edged blue screen effects. For example, the mask may be used to indicate which portions of the rendered image should be masked (and how much). This mask could be used by the graphics system or external hardware to blend the rendered image with another image (e.g., a signal from a video camera) to create a blue screen effect that is smooth (anti-aliased with respect to the overlapping regions of the two images) or a ghost effect (e.g., superimposing a partially transparent object smoothly over another object, scene, or video stream).

Example Output Pixel Calculation—FIG. 20

FIG. **20** illustrates a simplified example of an output pixel convolution. As the figure shows, four bins **288**A–D contain samples that may possibly contribute to the output pixel. In this example, the center of the output pixel is located at the boundary of bins **288**A–**288**D. Each bin comprises sixteen samples, and an array of four bins (2×2) is filtered to generate the output pixel. Assuming circular filters are used, the distance of each sample from the pixel center determines which filter value will be applied to the sample. For example, sample **296** is relatively close to the pixel center, and thus falls within the region of the filter having a filter value of 8. Similarly, samples **294** and **292** fall within the regions of the filter having filter values of 4 and 2, respectively. Sample **290**, however, falls outside the maximum filter extent, and thus receives a filter value of 0. Thus sample **290** will not contribute to the output pixel's value. This type of filter ensures that the samples located the closest to the pixel center will contribute the most, while pixels located farther from the pixel center will contribute less to the final output pixel values. This type of filtering automatically performs anti-aliasing by smoothing any abrupt changes in the image (e.g., from a dark line to a light background). Another particularly useful type of filter for anti-aliasing is a windowed sinc filter. Advantageously, the windowed sinc filter contains negative lobes that resharpen some of the blended or "fuzzed" image. Negative lobes are areas where the filter causes the samples to subtract from the pixel being calculated. In contrast samples on either side of the negative lobe add to the pixel being calculated.

Example values for samples **290**–**296** are illustrated in boxes **300**–**308**. In this example, each sample comprises red, green, blue and alpha values, in addition to the sample's positional data. Block **310** illustrates the calculation of each pixel component value for the non-normalized output pixel. As block **310** indicates, potentially undesirable gain is introduced into the final pixel values (i.e., an output pixel having a red component value of 2000 is much higher than any of the sample's red component values). As previously noted, the filter values may be summed to obtain normalization value **308**. Normalization value **308** is used to divide out the unwanted gain from the output pixel. Block **312** illustrates this process and the final normalized example pixel values.

Note the values used herein were chosen for descriptive purposes only and are not meant to be limiting. For example, the filter may have a large number of regions each with a

different filter value. In one embodiment, some regions may have negative filter values. The filter utilized may be a continuous function that is evaluated for each sample based on the sample's distance from the pixel center. Also note that floating point values may be used for increased precision. A variety of filters may be utilized, e.g., box, tent, cylinder, cone, Gaussian, Catmull-Rom, Mitchell and Netravalli, windowed sinc, etc.

It is also noted that the filter weights need not be powers of two as show in the figure. The example in the figure is simplified for explanatory purposes. A table of filter weights may be used (e.g., having a large number of entries which are indexed in to based on the distance of the sample from the pixel or filter center). Furthermore, in some embodiments each sample in each bin may be summed to form the pixel value (although some samples within the bins may have a weighting of zero and thus nevertheless contribute nothing to the final pixel value).

Full-Screen Anti-aliasing

The vast majority of current 3D graphics systems only provide real-time anti-aliasing for lines and dots. While some systems also allow the edge of a polygon to be "fuzzed", this technique typically works best when all polygons have been pre-sorted in depth. This may defeat the purpose of having general-purpose 3D rendering hardware for most applications (which do not depth pre-sort their polygons). In one embodiment, graphics system **112** may be configured to implement full-screen anti-aliasing by stochastically sampling up to sixteen samples per output pixel, filtered by a 5×5-convolution filter.

Variable-Resolution Super Sampling—FIGS. 21–25

Turning now to FIG. **21**, a diagram of one possible scheme for dividing sample buffer **162** is shown. In this embodiment, sample buffer **162** is divided into the following three nested regions: foveal region **354**, medial region **352**, and peripheral region **350**. Each of these regions has a rectangular shaped outer border, but the medial and the peripheral regions have a rectangular shaped hole in their center. Each region may be configured with certain constant (per frame) properties, e.g., a constant density sample density and a constant size of pixel bin. In one embodiment, the total density range may be 256, i.e., a region could support between one sample every 16 screen pixels (4×4) and 16 samples for every 1 screen pixel. In other embodiments, the total density range may be limited to other values, e.g., 64. In one embodiment, the sample density varies, either linearly or non-linearly, across a respective region. Note in other embodiments the display may be divided into a plurality of constant sized regions (e.g., squares that are 4×4 pixels in size or 40×40 pixels in size).

To simply perform calculations for polygons that encompass one or more region corners (e.g., a foveal region corner), the sample buffer may be further divided into a plurality of subregions. In FIG. **21**, one embodiment of sample buffer **162** divided into sub-regions is shown. Each of these sub-regions are rectangular, allowing graphics system **112** to translate from a 2D address with a sub-region to a linear address in sample buffer **162**. Thus, in some embodiments each sub-region has a memory base address, indicating where storage for the pixels within the sub-region starts. Each sub-region may also have a "stride" parameter associated with its width.

Another potential division of the super-sampled sample buffer is circular. Turning now to FIG. **22**, one such embodiment is illustrated. For example, each region may have two radii associated with it (i.e., **360–368**), dividing the region into three concentric circular-regions. The circular-regions may all be centered at the same screen point, the fovea center point. Note however, that the fovea center-point need not always be located at the center of the foveal region. In some instances it may even be located off-screen (i.e., to the side of the visual display surface of the display device). While the embodiment illustrated supports up to seven distinct circular-regions, it is possible for some of the circles to be shared across two different regions, thereby reducing the distinct circular-regions to five or less.

The circular regions may delineate areas of constant sample density actually used. For example, in the example illustrated in the figure, foveal region **354** may allocate a sample buffer density of 8 samples per screen pixel, but outside the innermost circle **368**, it may only use 4 samples per pixel, and outside the next circle **366** it may only use two samples per pixel. Thus, in this embodiment the rings need not necessarily save actual memory (the regions do that), but they may potentially save memory bandwidth into and out of the sample buffer (as well as pixel convolution bandwidth). In addition to indicating a different effective sample density, the rings may also be used to indicate a different sample position scheme to be employed. As previously noted, these sample position schemes may stored in an on-chip RAM/ROM, or in programmable memory.

As previously discussed, in some embodiments super-sampled sample buffer **162** may be further divided into bins. For example, a bin may store a single sample or an array of samples (e.g., 2×2 or 4×4 samples). In one embodiment, each bin may store between one and sixteen sample points, although other configurations are possible and contemplated. Each region may be configured with a particular bin size, and a constant memory sample density as well. Note that the lower density regions need not necessarily have larger bin sizes. In one embodiment, the regions (or at least the inner regions) are exact integer multiples of the bin size enclosing the region. This may allow for more efficient utilization of the sample buffer in some embodiments.

Variable-resolution super-sampling involves calculating a variable number of samples for each pixel displayed on the display device. Certain areas of an image may benefit from a greater number of samples (e.g., near object edges), while other areas may not need extra samples (e.g., smooth areas having a constant color and brightness). To save memory and bandwidth, extra samples may be used only in areas that may benefit from the increased resolution. For example, if part of the display is colored a constant color of blue (e.g., as in a background), then extra samples may not be particularly useful because they will all simply have the constant value (equal to the background color being displayed). In contrast, if a second area on the screen is displaying a 3D rendered object with complex textures and edges, the use of additional samples may be useful in avoiding certain artifacts such as aliasing. A number of different methods may be used to determine or predict which areas of an image would benefit from higher sample densities. For example, an edge analysis could be performed on the final image, and with that information being used to predict how the sample densities should be distributed. The software application may also be able to indicate which areas of a frame should be allocated higher sample densities.

A number of different methods may be used to implement variable-resolution super sampling. These methods tend to fall into the following two general categories:

(1) those methods that concern the draw or rendering process, and (2) those methods that concern the convolution process. For example, samples may be rendered into the super-sampling sample buffer **162** using any of the following methods:

1) a uniform sample density;
2) varying sample density on a per-region basis (e.g., medial, foveal, and peripheral); and
3) varying sample density by changing density on a scan-line basis (or on a small number of scan lines basis).

Varying sample density on a scan-line basis may be accomplished by using a look-up table of densities. For example, the table may specify that the first five pixels of a particular scan line have three samples each, while the next four pixels have two samples each, and so on.

On the convolution side, the following methods are possible:

1) a uniform convolution filter;
2) continuously variable convolution filter; and
3) a convolution filter operating at multiple spatial frequencies.

A uniform convolve filter may, for example, have a constant extent (or number of samples selected) for each pixel calculated. In contrast, a continuously variable convolution filter may gradually change the number of samples used to calculate a pixel. The function may be vary continuously from a maximum at the center of attention to a minimum in peripheral areas.

Different combinations of these methods (both on the rendering side and convolution side) are also possible. For example, a constant sample density may be used on the rendering side, while a continuously variable convolution filter may be used on the samples.

Different methods for determining which areas of the image will be allocated more samples per pixel are also contemplated. In one embodiment, if the image on the screen has a main focal point (e.g., a character like Mario in a computer game), then more samples may be calculated for the area around Mario and fewer samples may be calculated for pixels in other areas (e.g., around the background or near the edges of the screen).

In another embodiment, the viewer's point of foveation may be determined by eye/head/hand-tracking. In head-tracking embodiments, the direction of the viewer's gaze is determined or estimated from the orientation of the viewer's head, which may be measured using a variety of mechanisms. For example, a helmet or visor worn by the viewer (with eye/head tracking) may be used alone or in combination with a hand-tracking mechanism, wand, or eye-tracking sensor to provide orientation information to graphics system **112**. Other alternatives include head-tracking using an infrared reflective dot placed on the user's forehead, or using a pair of glasses with head- and or eye-tracking sensors built in. One method for using head- and hand-tracking is disclosed in U.S. Pat. No. 5,446,834 (entitled "Method and Apparatus for High Resolution Virtual Reality Systems Using Head Tracked Display," by Michael Deering, issued Aug. 29, 1995), which is incorporated herein by reference in its entirety. Other methods for head tracking are also possible and contemplated (e.g., infrared sensors, electromagnetic sensors, capacitive sensors, video cameras, sonic and ultrasonic detectors, clothing based sensors, video tracking devices, conductive ink, strain gauges, force-feedback detectors, fiber optic sensors, pneumatic sensors, magnetic tracking devices, and mechanical switches).

As previously noted, eye-tracking may be particularly advantageous when used in conjunction with head-tracking. In eye-tracked embodiments, the direction of the viewer's gaze is measured directly by detecting the orientation of the viewer's eyes in relation to the viewer's head. This information, when combined with other information regarding the position and orientation of the viewer's head in relation to the display device, m ay allow an accurate measurement of viewer's point of foveation (or points of foveation if two eye-tracking sensors are used). One possible method for eye tracking is disclosed in U.S. Pat. No. 5,638,176 (entitled "Inexpensive Interferometric Eye Tracking System"). Other methods for eye tracking are also possible and contemplated (e.g., the methods for head tracking listed above).

Regardless of which method is used, as the viewer's point of foveation changes position, so does the distribution of samples. For example, if the viewer's gaze is focused on the upper left-hand corner of the screen, the pixels corresponding to the upper left-hand corner of the screen may each be allocated eight or sixteen samples, while the pixels in the opposite corner (i.e., the lower right-hand corner of the screen) may be allocated only one or two samples per pixel. Once the viewer's gaze changes, so does the allotment of samples per pixel. When the viewer's gaze moves to the lower right-hand corner of the screen, the pixels in the upper left-hand corner of the screen may be allocated only one or two samples per pixel. Thus the number of samples per pixel may be actively changed for different regions of the screen in relation the viewers point of foveation. Note in some embodiments, multiple users may be each have head/eye/hand tracking mechanisms that provide input to graphics system **112**. In these embodiments, there may conceivably be two or more points of foveation on the screen, with corresponding areas of high and low sample densities. As previously noted, these sample densities may affect the render process only, the filter process only, or both processes.

Turning now to FIGS. **24**A–B, one embodiment of a method for apportioning the number of samples per pixel is shown. The method apportions the number of samples based on the location of the pixel relative to one or more points of foveation. In FIG. **24**A, an eye- or head-tracking device **360** is used to determine the point of foveation **362** (i.e., the focal point of a viewer's gaze). This may be determined by using tracking device **360** to determine the direction that the viewer's eyes (represented as **364** in the figure) are facing. As the figure illustrates, in this embodiment, the pixels are divided into foveal region **354** (which may be centered around the point of foveation **362**), medial region **352**, and peripheral region **350**.

Three sample pixels are indicated in the figure. Sample pixel **374** is located within foveal region **314**. Assuming foveal region **314** is configured with bins having eight samples, and assuming the convolution radius for each pixel touches four bins, then a maximum of 32 samples may contribute to each pixel. Sample pixel **372** is located within medial region **352**. Assuming medial region **352** is configured with bins having four samples, and assuming the convolution radius for each pixel touches four bins, then a maximum of 16 samples may contribute to each pixel. Sample pixel **370** is located within peripheral region **350**. Assuming peripheral region **370** is configured with bins having one sample each, and assuming the convolution radius for each pixel touches one bin, then there is a one sample to pixel correlation for pixels in peripheral region **350**. Note these values are merely examples and a different number of regions, samples per bin, and convolution radius may be used.

Turning now to FIG. **24**B, the same example is shown, but with a different point of foveation **362**. As the figure illustrates, when tracking device **360** detects a change in the position of point of foveation **362**, it provides input to the graphics system, which then adjusts the position of foveal region **354** and medial region **352**. In some embodiments, parts of some of the regions (e.g., medial region **352**) may extend beyond the edge of display device **84**. In this example, pixel **370** is now within foveal region **354**, while pixels **372** and **374** are now within the peripheral region. Assuming the sample configuration as the example in FIG. **24**A, a maximum of 32 samples may contribute to pixel **370**, while only one sample will contribute to pixels **372** and **374**. Advantageously, this configuration may allocate more samples for regions that are near the point of foveation (i.e., the focal point of the viewer's gaze). This may provide a more realistic image to the viewer without the need to calculate a large number of samples for every pixel on display device **84**.

Turning now to FIGS. **25**A–B, another embodiment of a computer system configured with a variable resolution super-sampled sample buffer is shown. In this embodiment, the center of the viewer's attention is determined by position of a main character **362**. Medial and foveal regions are centered around main character **362** as it moves around the screen. In some embodiments main character may be a simple cursor (e.g., as moved by keyboard input or by a mouse).

In still another embodiment, regions with higher sample density may be centered around the middle of display device **84**'s screen. Advantageously, this may require less control software and hardware while still providing a shaper image in the center of the screen (where the viewer's attention may be focused the majority of the time).

Although the embodiments above have been described in considerable detail, other versions are possible. Numerous variations and modifications will become apparent to those skilled in the art once the above disclosure is fully appreciated. It is intended that the following claims be interpreted to embrace all such variations and modifications. Note the headings used herein are for organizational purposes only and are not meant to limit the description provided herein or the claims attached hereto.

What is claimed is:

1. A graphics system comprising:

one or more processors configured to receive a set of three-dimensional graphics data and render a plurality of samples based on the graphics data;

a sample buffer configured to store the plurality of samples; and

a plurality of sample-to-pixel calculation unit, wherein the sample-to-pixel calculation units are configured to receive and filter samples from the sample buffer to create output pixels, wherein the output pixels are usable to form an image on a display device, wherein each of the plurality of sample-to-pixel calculation units are configured to generate pixels corresponding to a different one of a plurality of regions of the image.

2. The graphics system as recited in claim **1**, wherein the processors are configured to receive the three-dimensional graphics data in a compressed form, and wherein the processors are configured to decompress the three-dimensional graphics data before rendering the samples.

3. The graphics system as recited in claim **1**, wherein each region corresponds to a different vertical stripe of the image.

4. The graphics system as recited in claim **1**, wherein each region comprises portions of the image that correspond to one or more odd or even scan lines.

5. The graphics system as recited in claim **1**, wherein each region comprises a different quadrant of the image.

6. The graphics system as recited in claim 1, wherein the plurality of regions overlap.

7. The graphics system as recited in claim 1, wherein the display device comprises a plurality of individual display devices, and wherein each region corresponds to a single one of the plurality of individual display devices.

8. The graphics system as recited in claim 1, wherein the display device comprises a plurality of individual display devices, and wherein each region corresponds to a different one of the plurality of individual display devices.

9. The graphics system as recited in claim 1, wherein the regions vary in dimension on a frame-by-frame basis.

10. The graphics system as recited in claim 1, wherein the regions of the image vary in dimension on a frame-by-frame basis to balance the number of samples filtered by each of the sample-to-pixel calculation units.

11. The graphics system as recited in claim 1, wherein each region is a different horizontal stripe of the image.

12. The graphics system as recited in claim 1, wherein each region is a different rectangular portion of the image.

13. The graphics system as recited in claim 1, wherein each sample comprises color components, and wherein the sample-to-pixel calculation units are configured to:

determine which samples are within a predetermined filter envelope;

multiply the samples within the predetermined filter envelope by one or more un-normalized weighting factors, wherein the weighting factors vary in relation to the sample's position relative to the center of the filter envelope; and

normalize the resulting output pixels.

14. The graphics system as recited in claim 1, wherein each sample comprises color components, and wherein the sample-to-pixel calculation units are configured to determine which samples are within a predetermined filter envelope, and multiply the samples within the predetermined filter envelope by one or more normalized weighting factors, wherein the weighting factors vary in relation to the sample's position relative to the center of the filter envelope.

15. The graphics system as recited in claim 1, wherein each sample comprises an alpha component.

16. The graphics system as recited in claim 1, wherein each sample comprises a blur component.

17. The graphics system as recited in claim 1, wherein each sample comprises a transparency component.

18. The graphics system as recited in claim 1, wherein each sample comprises a z-component.

19. The graphics system as recited in claim 1, wherein the samples stored in the sample buffer are double buffered.

20. The graphics system as recited in claim 1, wherein the samples stored in the sample buffer are stored in bins.

21. The graphics system as recited in claim 1, further comprising the display device.

22. A method for rendering a set of three-dimensional graphics data, the method comprising:

receiving the three-dimensional graphics data;

generating one or more samples based on the graphics data;

storing the samples;

dividing the samples into a plurality of regions;

selecting stored samples from the plurality of regions; and

filtering the selected samples to form a plurality of output pixels in parallel, wherein the output pixels are usable to form an image on a display device.

23. The method as recited in claim 22, wherein the plurality of regions comprise portions of one or more odd or even scan lines of the image.

24. The method as recited in claim 22, wherein each region comprises a quadrant of the image.

25. The method as recited in claim 22, wherein each region comprises a vertical stripe of the image.

26. The method as recited in claim 22, wherein each region corresponds to a horizontal stripe of the image.

27. The method as recited in claim 22, wherein the display device comprises a plurality of individual display devices, and wherein each region corresponds to a single one of the plurality of individual display devices.

28. The method as recited in claim 22, wherein the display device comprises a plurality of individual display devices, and wherein each region corresponds to a different one of the plurality of individual display devices.

29. The method as recited in claim 22, wherein each region comprises a vertical stripe of the image.

30. The method as recited in claim 22, wherein the regions of the image overlap.

31. The method as recited in claim 22, wherein the boundaries of the regions change over time to balance the number of samples in each vertical column.

32. The method as recited in claim 22, wherein the number of samples filtered per pixel varies across the image.

33. The method as recited in claim 22, wherein the boundaries of the regions change over time to equalized the number of samples in each region.

34. The method as recited in claim 22, wherein the three-dimensional graphics data is received in compressed form, and wherein the method further comprises: decompressing the compressed three-dimensional graphics data.

35. The method as recited in claim 22, wherein each sample comprises color components, and wherein said filtering comprises:

determining which samples are within a predetermined filter envelope;

multiplying the samples within the predetermined filter envelope by one or more un-normalized weighting factors, wherein said weighting factors vary in relation to the sample's position relative to the center of the filter envelope;

summing the weighted samples to form an output pixel; and

normalizing the output pixel.

36. The method as recited in claim 22, wherein each sample comprises color components, and wherein said filtering comprises:

determining which samples are within a predetermined filter envelope;

multiplying the samples within the predetermined filter envelope by one or more normalized weighting factors, wherein the weighting factors vary in relation to the sample's position relative to the center of the filter envelope; and

summing the weighted samples to form an output pixel.

37. The method as recited in claim 22, wherein the samples are stored in bins.

38. A computer system comprising:

a means for receiving a set of three-dimensional graphics data;

a means for rendering a plurality of samples based on the set of three-dimensional graphics data;

a means for storing the rendered samples; and

a plurality of filtering means configured to filter stored samples to create output pixels, wherein the output pixels are usable to form an image on a display device, and wherein each of the plurality of filtering means are configured to generate pixels corresponding to one of a plurality of different regions of the image.

39. The computer system as recited in claim **38**, wherein each region corresponds to a different vertical stripe of the image.

40. The computer system as recited in claim **38**, wherein the regions overlap.

41. The computer system as recited in claim **38**, wherein the regions vary in size over time.

42. The computer system as recited in claim **38**, wherein each region varies in size on a frame-by-frame basis to balance the number of samples filtered by each of the filtering means.

43. The computer system as recited in claim **38**, wherein each region corresponds to a different horizontal stripe of the image.

44. The computer system as recited in claim **38**, wherein each region corresponds to a different rectangular portion of the image.

45. The computer system as recited in claim **38**, wherein each region corresponds to a different quadrant of the image.

46. The method as recited in claim **38**, wherein one or more of the plurality of regions comprises portions of the image corresponding to one or more odd scan lines of the image, and wherein one or more of the plurality of regions comprises portions of the image corresponding to one or more even scan lines of the image.

47. The computer system as recited in claim **38**, wherein the size of each portion of the image varies on a frame-by-frame basis to balance the number of samples filtered by each of the filtering means.

48. The computer system as recited in claim **38**, wherein each sample comprises color components, and wherein the filtering means are configured to determine which samples are within a predetermined filter envelope, multiply the samples within the predetermined filter envelope by one or more un-normalized weighting factors, wherein the weighting factors varies in relation to the sample's position relative to the center of the filter envelope, and normalize the resulting output pixels.

49. The computer system as recited in claim **38**, wherein each sample comprises color components, and wherein the filtering means are configured to determine which samples are within a predetermined filter envelope, multiply the samples within the predetermined filter envelope by one or more normalized weighting factors to form one or more output pixels.

50. The computer system as recited in claim **38**, wherein each sample further comprises an alpha component.

51. The computer system as recited in claim **38**, wherein the samples stored in the sample buffer are double buffered.

52. The computer system as recited in claim **38**, wherein the samples stored in the sample buffer are stored in bins.

* * * * *

US006876704B2

(12) **United States Patent**
LaRocca et al.

(10) **Patent No.:** **US 6,876,704 B2**
(45) **Date of Patent:** **Apr. 5, 2005**

(54) APPARATUS AND METHOD FOR ENCODING AND COMPUTING A DISCRETE COSINE TRANSFORM USING A BUTTERFLY PROCESSOR

(75) Inventors: **Judith LaRocca**, Carlsbad, CA (US); **A. Chris Irvine**, Bonsall, CA (US); **Jeffrey A. Levin**, San Diego, CA (US)

(73) Assignee: **Qualcomm, Incorporated**, San Diego, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 849 days.

(21) Appl. No.: **09/876,789**

(22) Filed: **Jun. 6, 2001**

(65) **Prior Publication Data**

US 2002/0181027 A1 Dec. 5, 2002

Related U.S. Application Data

(60) Provisional application No. 60/291,467, filed on May 16, 2001.

(51) **Int. Cl.**^{7} **H04N 7/12**
(52) **U.S. Cl.** **375/240.2**
(58) **Field of Search** 375/240.2, 240.21, 375/240.24, 240.23; 364/725.01, 725, 726, 727; 382/248, 250; 348/395.1, 408.1; H04N 7/12

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,592,399	A	*	1/1997	Keith et al.	709/247
5,610,849	A		3/1997	Huang	
5,684,534	A	*	11/1997	Harney et al.	375/240.25
5,818,742	A		10/1998	Fraenkel et al.	
6,366,585	B1	*	4/2002	Dapper et al.	370/409
6,397,240	B1	*	5/2002	Fernando et al.	708/603
6,687,315	B2	*	2/2004	Keevill et al.	375/341

FOREIGN PATENT DOCUMENTS

EP	0424119	4/1991
EP	0566184	10/1993
EP	0714212	5/1996

OTHER PUBLICATIONS

Vaisey J. et al., "Image Compression with Variable Block Size Segmentation" IEE Transactions on Signal Processing, IEEE, Inc. New York, US, vol. 40, No. 8, Aug. 1, 1992, pp. 2040–2060.

Chen R–J et al., "A Fully Adaptive DCT Based Color Image Sequence Coder" Signal Processing. Image Communication, Elsevier Science Publishers, Amsterdam, NL, vol. 6, No. 4, Aug. 1, 1994, pp. 289–301.

Kato Y et al., "An Adaptive Orthogonal Transform Coding Algorithm for Images Utilizing Classification Technique" Electronics & Communications in Japan, Part I—Communications, Scripta Technica. New York, US, vol. 72, No. 5, Part 1, May 1, 1989, pp. 1–9.

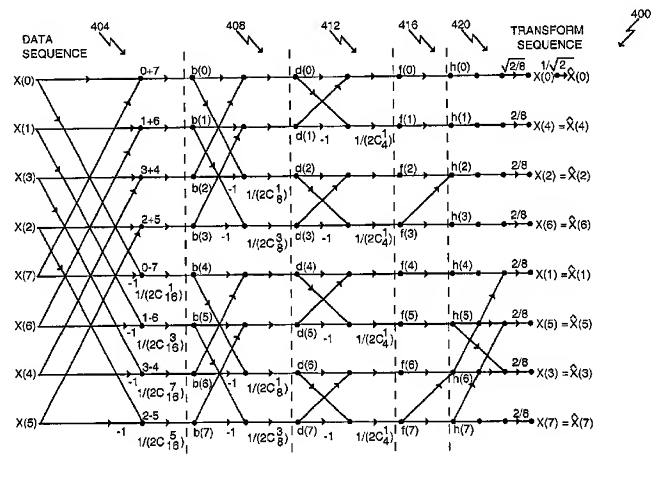
* cited by examiner

Primary Examiner—Nhon Diep
(74) *Attorney, Agent, or Firm*—Sandip (Micky) S. Minhas; Phil Wadsworth

(57) **ABSTRACT**

An apparatus to determine a transform of a block of encoded data the block of encoded data comprising a plurality of data elements. An input register is configured to receive a predetermined quantity of data elements. At least one butterfly processor is coupled to the input register and is configured to perform at least one mathematical operation on selected pairs of data elements to produce an output of processed data elements. At least one intermediate register is coupled to the butterfly processor and configured to temporarily store the processed data. A feedback loop is coupled to the intermediate register and the butterfly processor, and where if enabled, is configured to transfer a first portion of processed data elements to the appropriate butterfly processor to perform additional mathematical operations and where if disabled, is configured to transfer a second portion of processed data elements to at least one holding register.

91 Claims, 12 Drawing Sheets



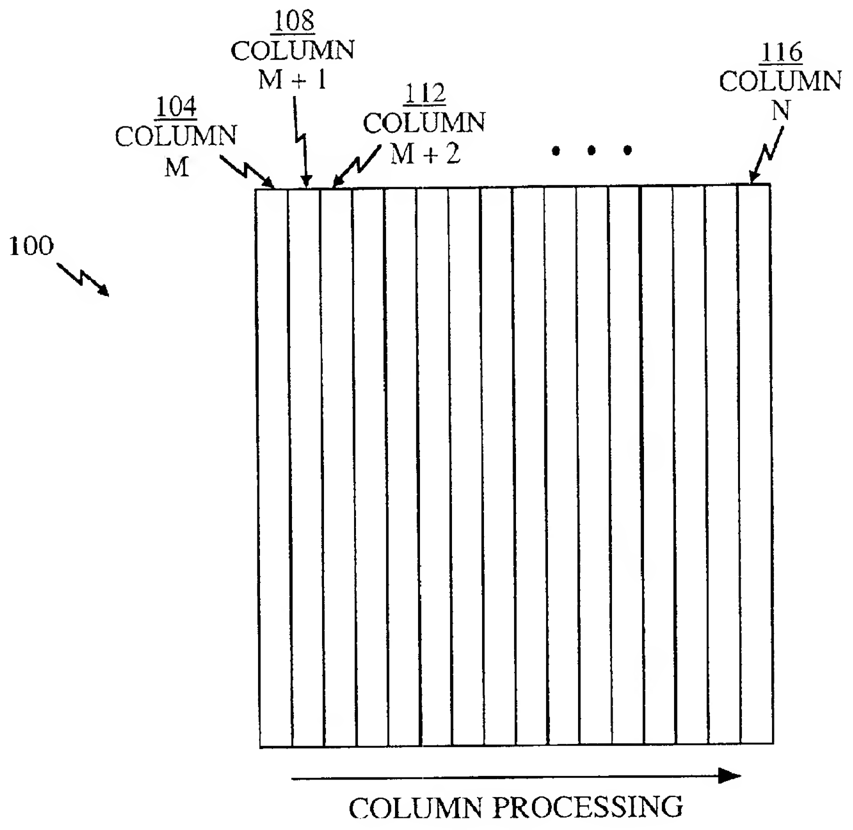

108
COLUMN
M + 1
104
COLUMN
M
112
COLUMN
M + 2
116
COLUMN
N
100
COLUMN PROCESSING

FIG. 1A

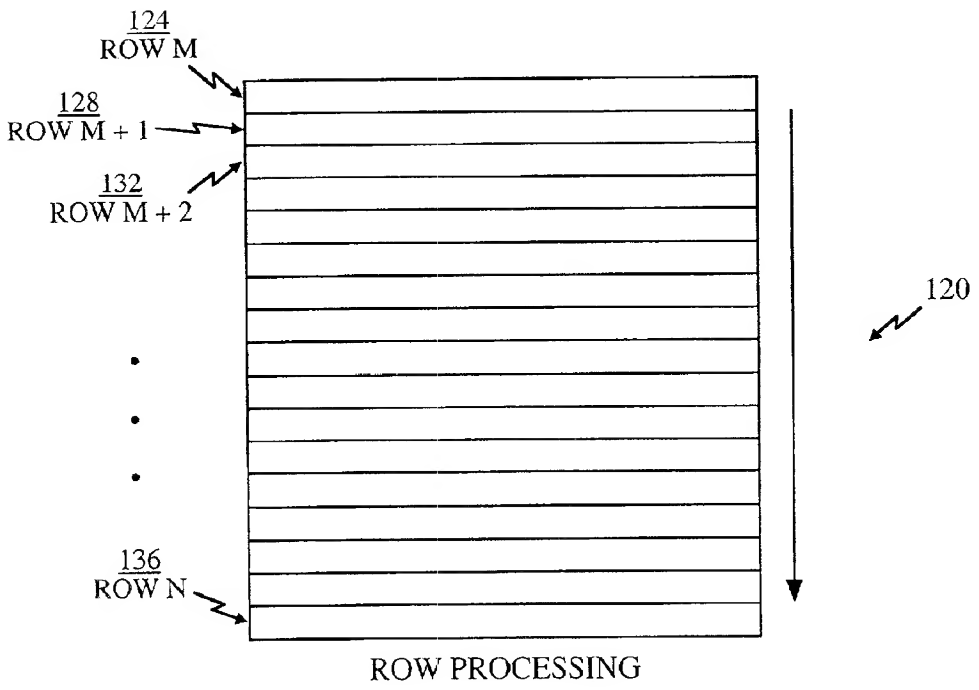

124
ROW M
128
ROW M + 1
132
ROW M + 2
136
ROW N
120
ROW PROCESSING

FIG. 1B

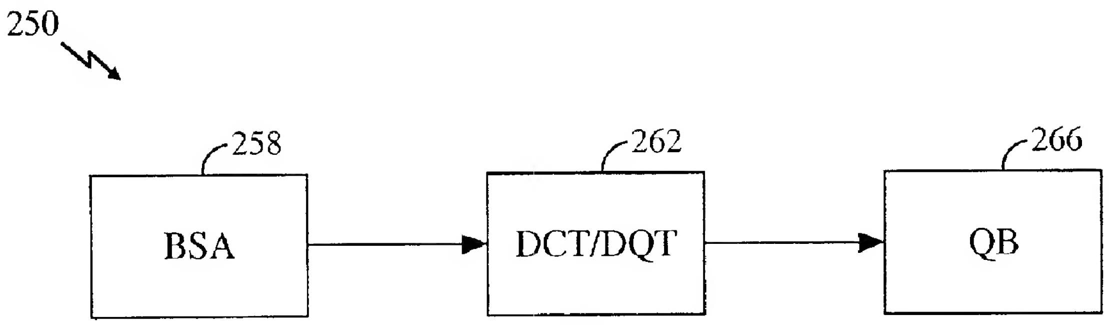


FIG. 2A

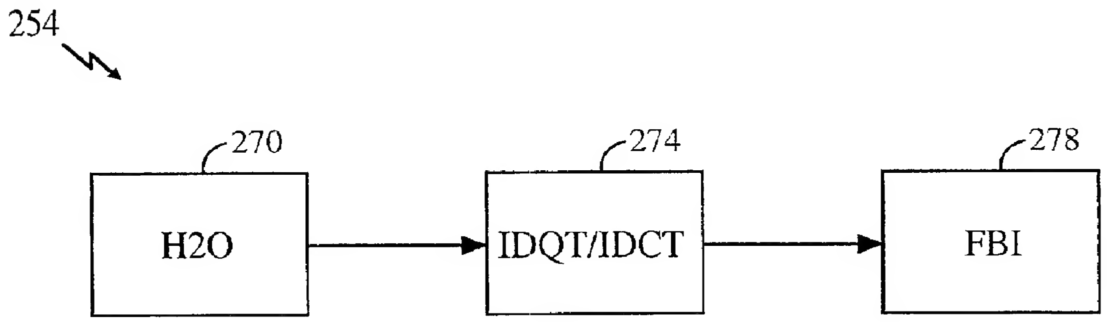


FIG. 2B

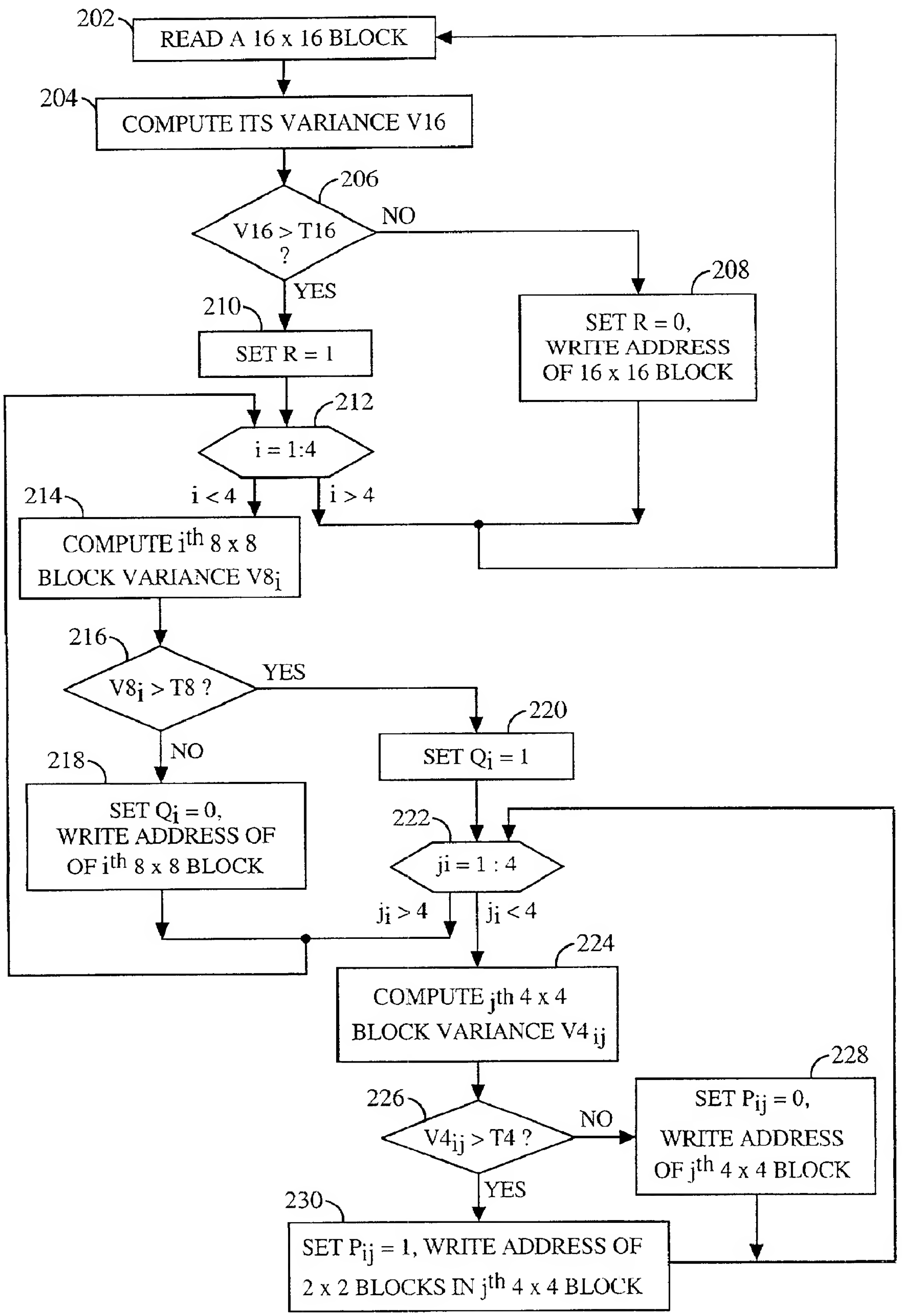

202
READ A 16 x 16 BLOCK
204
COMPUTE ITS VARIANCE V16
206
V16 > T16 ?
NO
YES
208
SET R = 0, WRITE ADDRESS OF 16 x 16 BLOCK
210
SET R = 1
212
i = 1:4
i < 4
i > 4
214
COMPUTE ith 8 x 8 BLOCK VARIANCE V8i
216
V8i > T8 ?
YES
NO
220
SET Qi = 1
218
SET Qi = 0, WRITE ADDRESS OF OF ith 8 x 8 BLOCK
222
ji = 1 : 4
ji > 4
ji < 4
224
COMPUTE jth 4 x 4 BLOCK VARIANCE V4ij
226
V4ij > T4 ?
NO
228
SET Pij = 0, WRITE ADDRESS OF jth 4 x 4 BLOCK
YES
230
SET Pij = 1, WRITE ADDRESS OF 2 x 2 BLOCKS IN jth 4 x 4 BLOCK

FIG. 2C

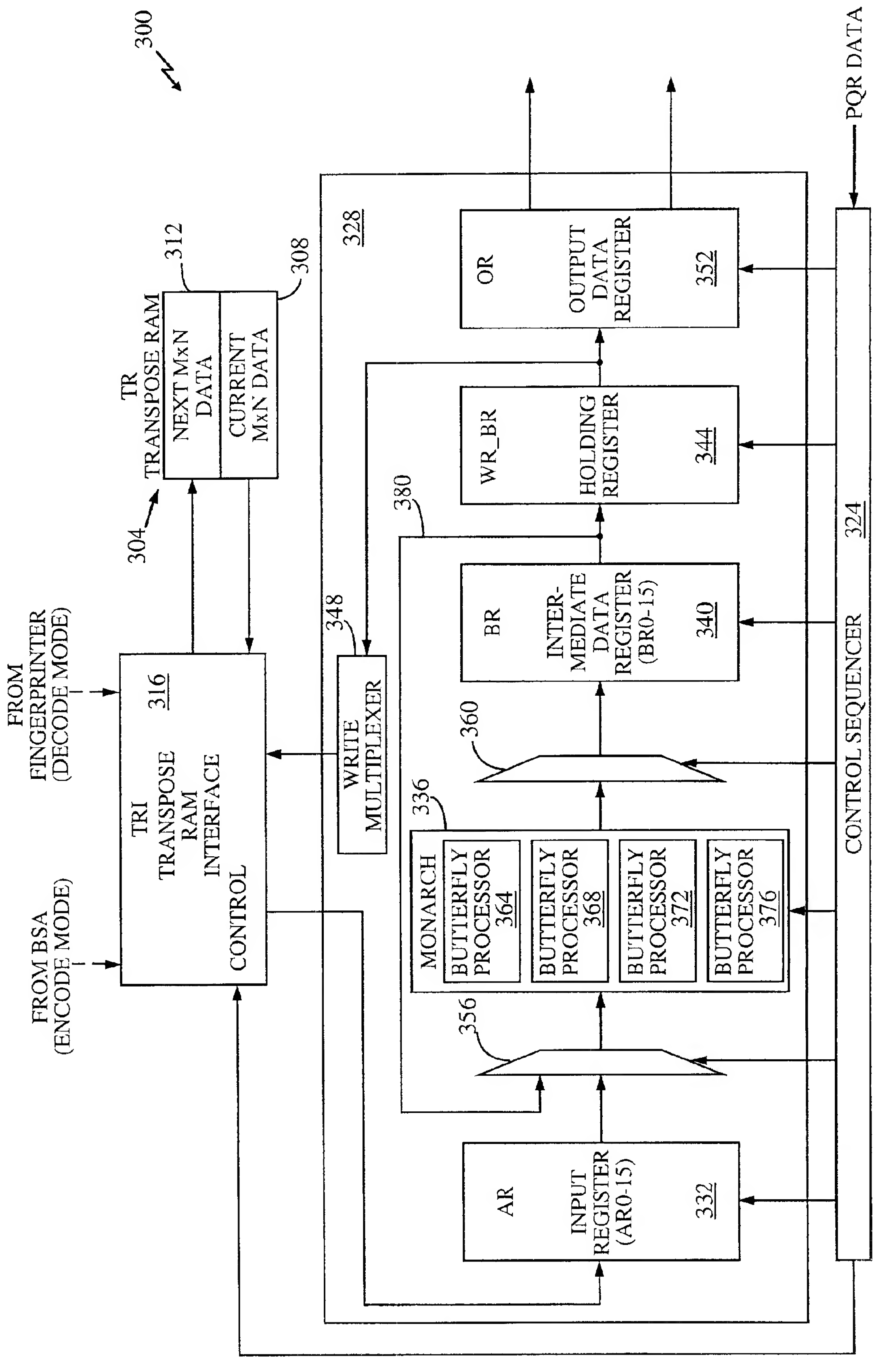

300
FROM BSA (ENCODE MODE)
FROM FINGERPRINTER (DECODE MODE)
TRI TRANSPOSE RAM INTERFACE CONTROL
316
304
TR TRANSPOSE RAM
312
NEXT MxN DATA
CURRENT MxN DATA
308
328
348
WRITE MULTIPLEXER
380
336
MONARCH
BUTTERFLY PROCESSOR 364
BUTTERFLY PROCESSOR 368
BUTTERFLY PROCESSOR 372
BUTTERFLY PROCESSOR 376
356
360
AR
INPUT REGISTER (AR0-15)
332
BR
INTER-MEDIATE DATA REGISTER (BR0-15)
340
WR_BR
HOLDING REGISTER
344
OR
OUTPUT DATA REGISTER
352
CONTROL SEQUENCER
324
PQR DATA

FIG. 3

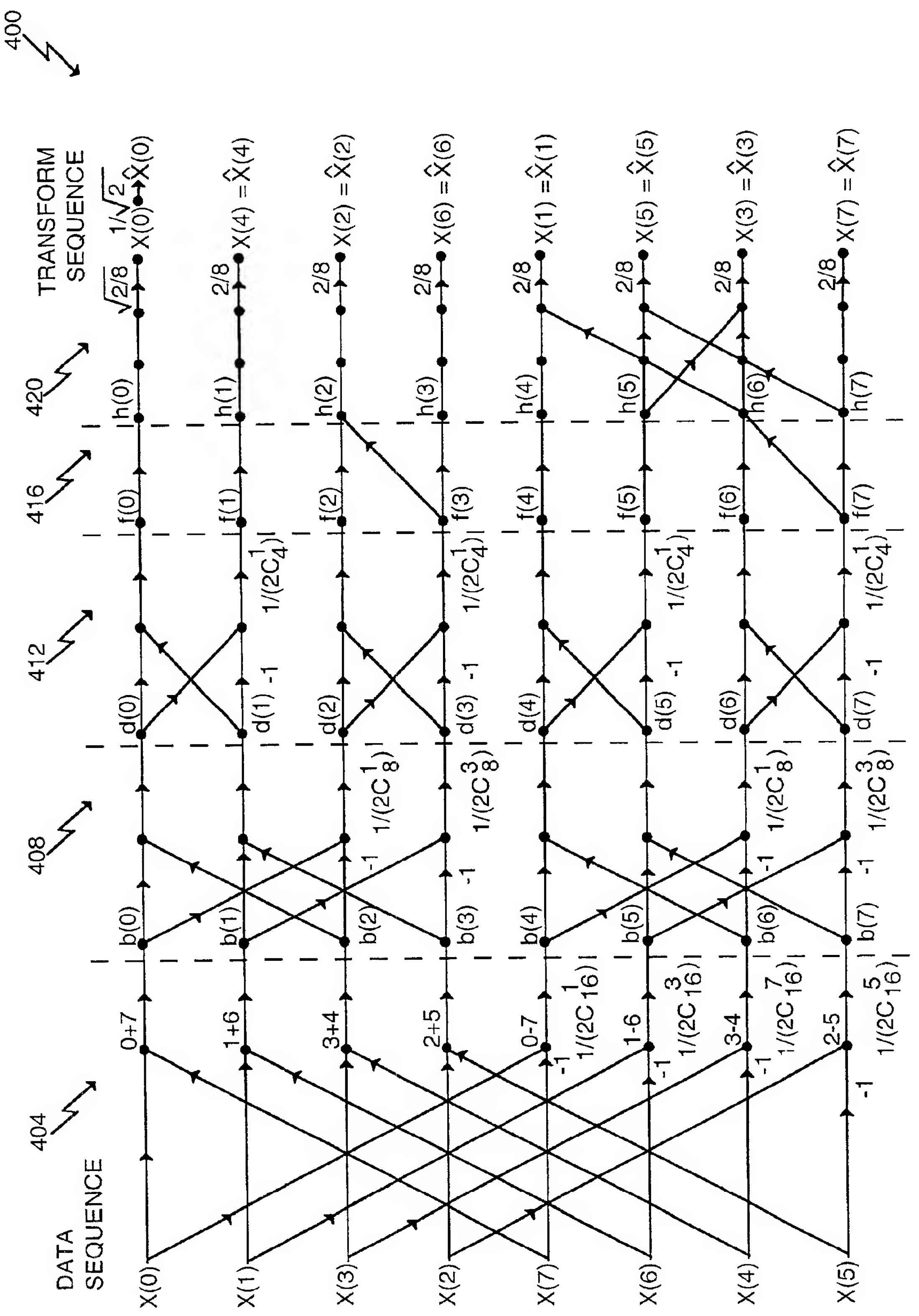

400
404
408
412
416
420
DATA SEQUENCE
X(0)
X(1)
X(3)
X(2)
X(7)
X(6)
X(4)
X(5)
0+7
1+6
3+4
2+5
0-7
1-6
3-4
2-5
-1
$1/(2C_{16}^1)$
$1/(2C_{16}^3)$
$1/(2C_{16}^7)$
$1/(2C_{16}^5)$
b(0)
b(1)
b(2)
b(3)
b(4)
b(5)
b(6)
b(7)
$1/(2C_8^1)$
$1/(2C_8^3)$
d(0)
d(1)
d(2)
d(3)
d(4)
d(5)
d(6)
d(7)
$1/(2C_4^1)$
f(0)
f(1)
f(2)
f(3)
f(4)
f(5)
f(6)
f(7)
h(0)
h(1)
h(2)
h(3)
h(4)
h(5)
h(6)
h(7)
$\sqrt{2/8}$
2/8
$1/\sqrt{2}$
TRANSFORM SEQUENCE
$X(0) \to \hat{X}(0)$
$X(4) = \hat{X}(4)$
$X(2) = \hat{X}(2)$
$X(6) = \hat{X}(6)$
$X(1) = \hat{X}(1)$
$X(5) = \hat{X}(5)$
$X(3) = \hat{X}(3)$
$X(7) = \hat{X}(7)$

FIG. 4

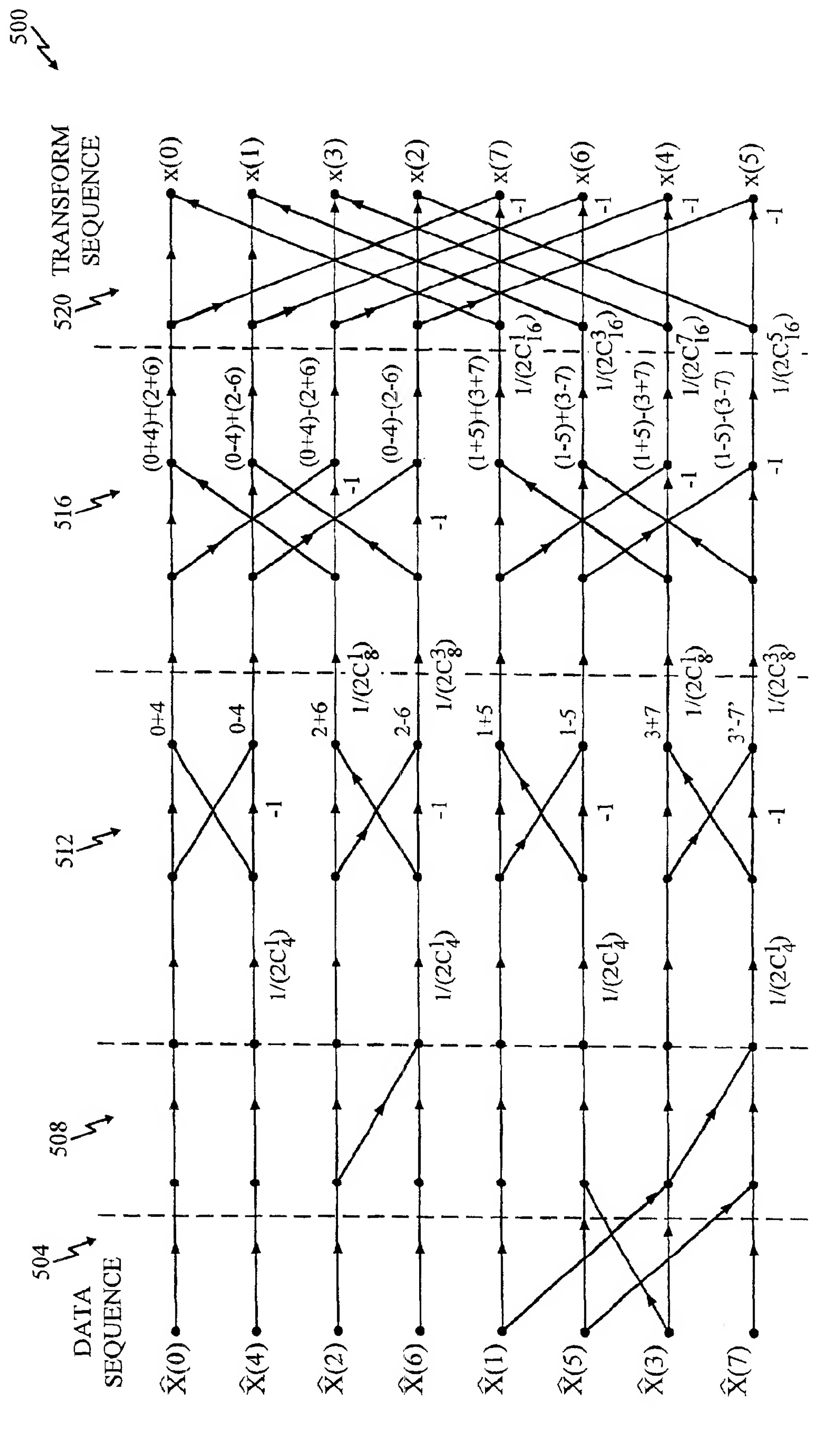

500
DATA SEQUENCE
$\hat{X}(0)$
$\hat{X}(4)$
$\hat{X}(2)$
$\hat{X}(6)$
$\hat{X}(1)$
$\hat{X}(5)$
$\hat{X}(3)$
$\hat{X}(7)$
504
508
512
516
520
TRANSFORM SEQUENCE
x(0)
x(1)
x(3)
x(2)
x(7)
x(6)
x(4)
x(5)
$1/(2C_4^1)$
-1
0+4
0-4
2+6
2-6
1+5
1-5
3+7
3'-7'
$1/(2C_8^1)$
$1/(2C_8^3)$
(0+4)+(2+6)
(0-4)+(2-6)
(0+4)-(2+6)
(0-4)-(2-6)
(1+5)+(3+7)
(1-5)+(3-7)
(1+5)-(3+7)
(1-5)-(3-7)
$1/(2C_{16}^1)$
$1/(2C_{16}^3)$
$1/(2C_{16}^7)$
$1/(2C_{16}^5)$

FIG. 5

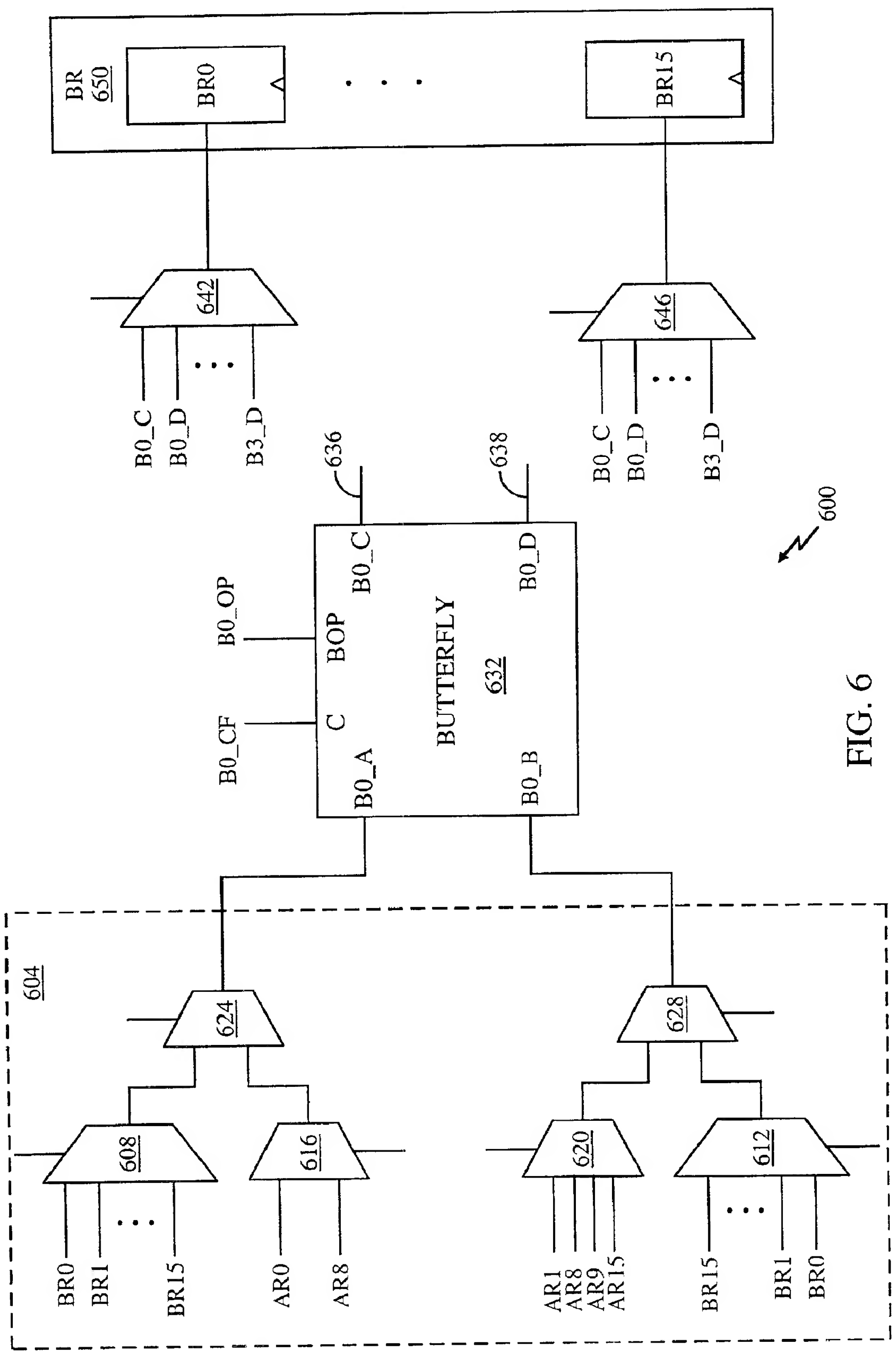

BR
650
BR0
BR15
642
646
B0_C
B0_D
B3_D
636
638
BUTTERFLY
632
B0_OP
BOP
B0_CF
C
B0_A
B0_B
B0_C
B0_D
600
604
624
628
608
616
620
612
BR0
BR1
BR15
AR0
AR8
AR1
AR8
AR9
AR15
BR15
BR1
BR0

FIG. 6

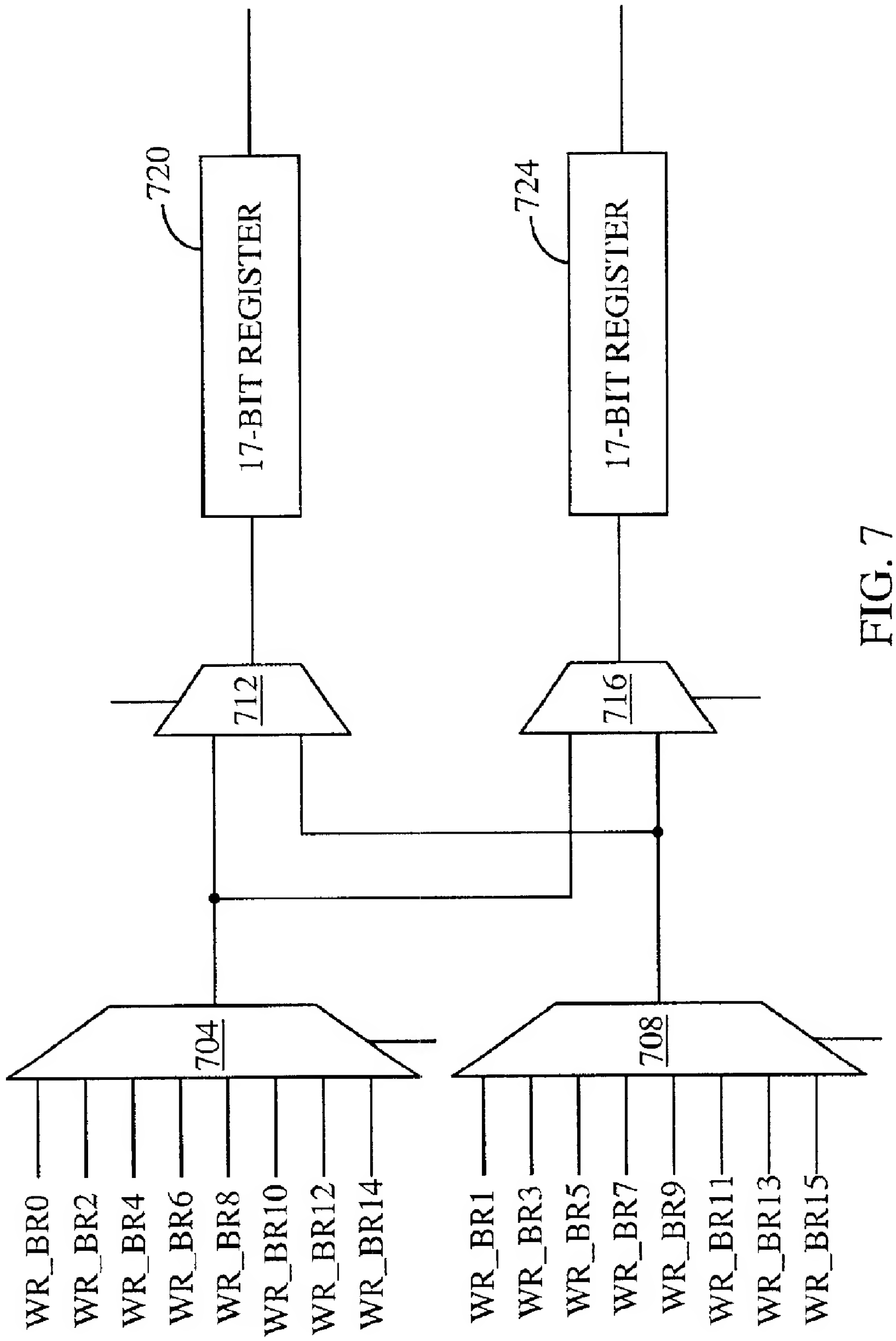

720
17-BIT REGISTER
724
17-BIT REGISTER
712
716
704
708
WR_BR0
WR_BR2
WR_BR4
WR_BR6
WR_BR8
WR_BR10
WR_BR12
WR_BR14
WR_BR1
WR_BR3
WR_BR5
WR_BR7
WR_BR9
WR_BR11
WR_BR13
WR_BR15

FIG. 7

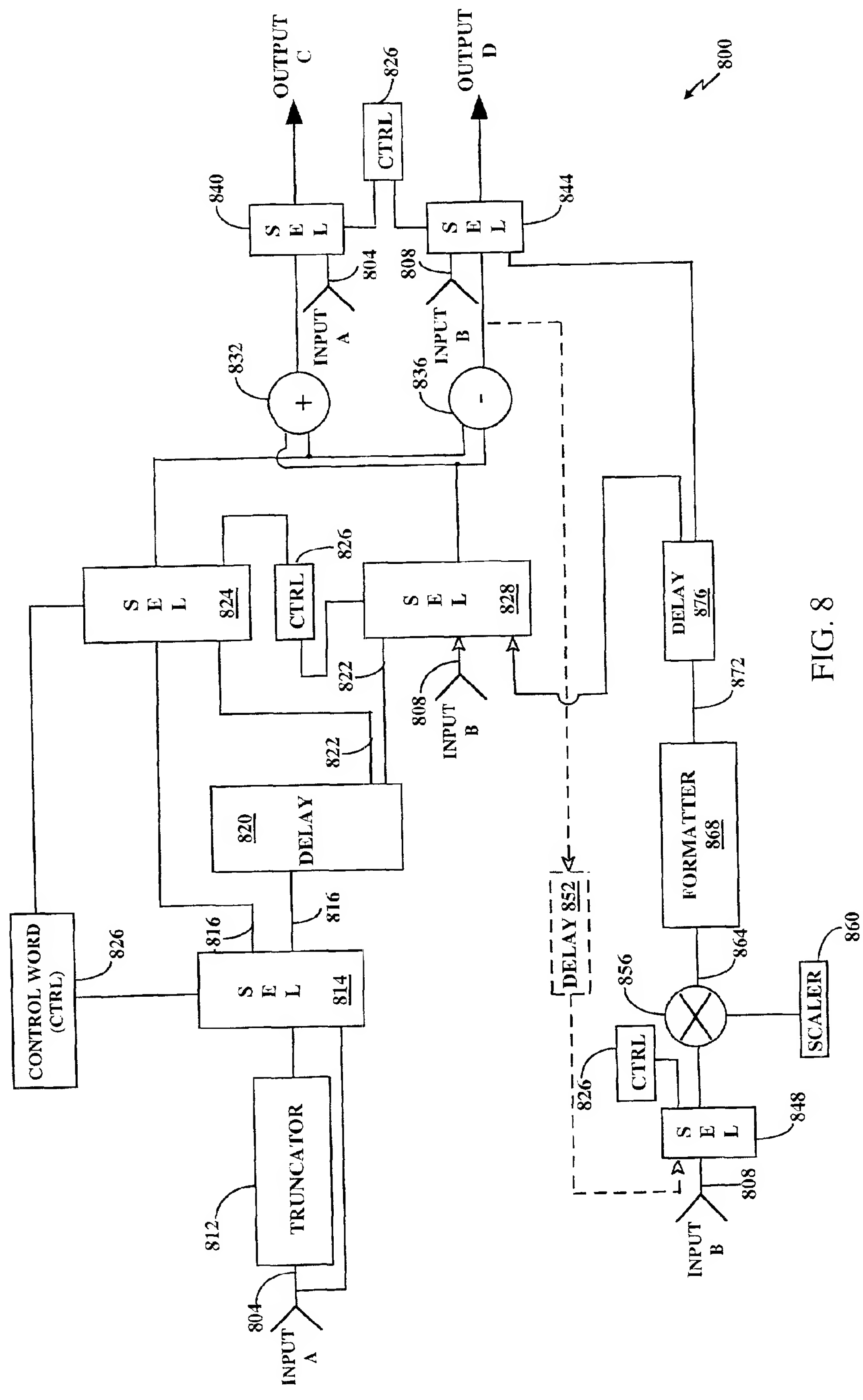

OUTPUT C
OUTPUT D
CTRL
826
800
840
844
804
808
INPUT A
INPUT B
832
836
+
-
S E L
824
CTRL
826
S E L
828
DELAY
876
822
822
808
INPUT B
820
DELAY
816
816
CONTROL WORD (CTRL)
826
S E L
814
TRUNCATOR
812
804
INPUT A
DELAY 852
FORMATTER
868
872
864
856
SCALER
860
CTRL
826
S E L
848
808
INPUT B

FIG. 8

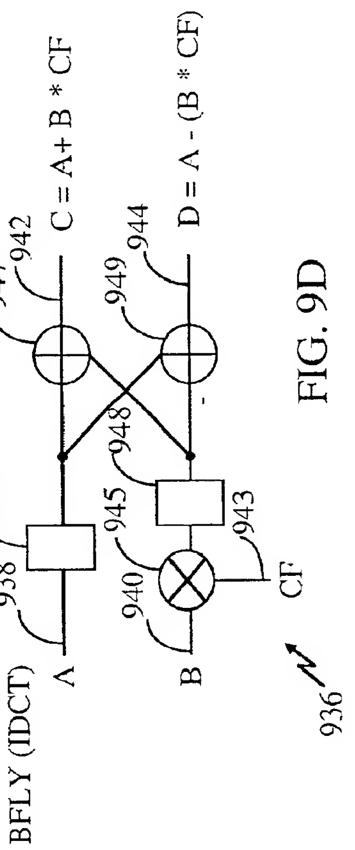


FIG. 9D

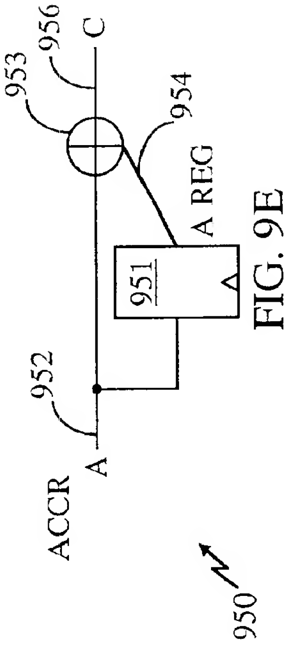


FIG. 9E

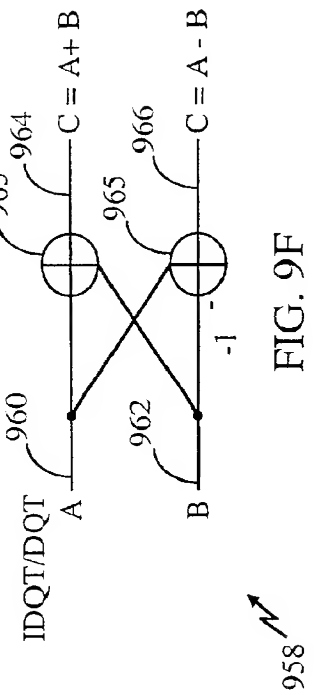


FIG. 9F

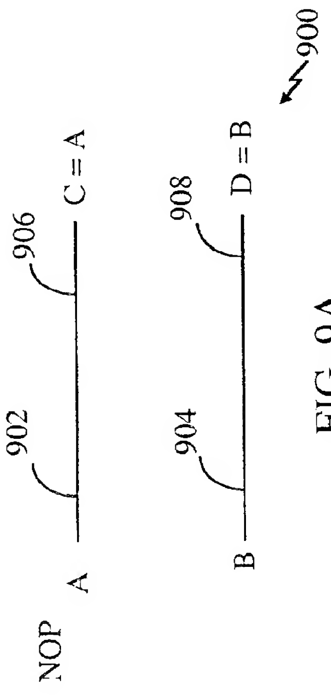


FIG. 9A

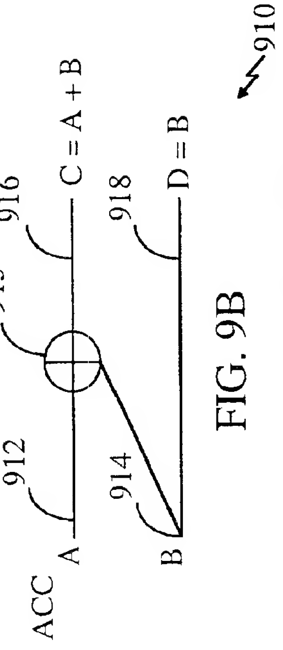


FIG. 9B

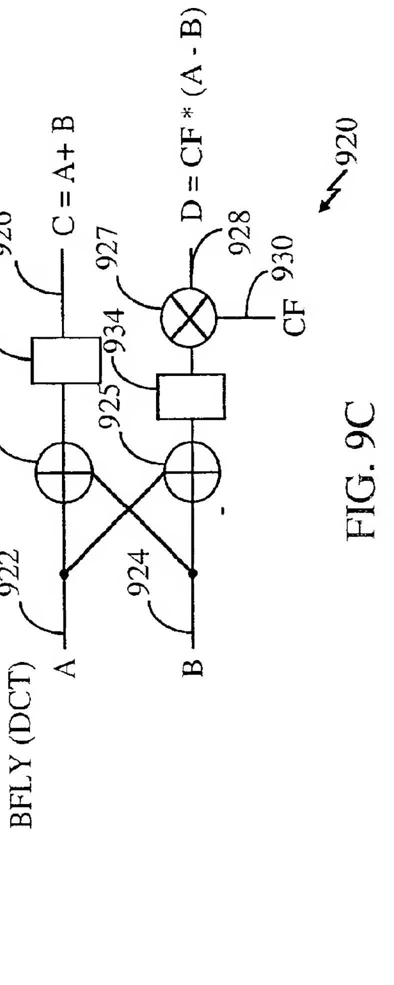


FIG. 9C

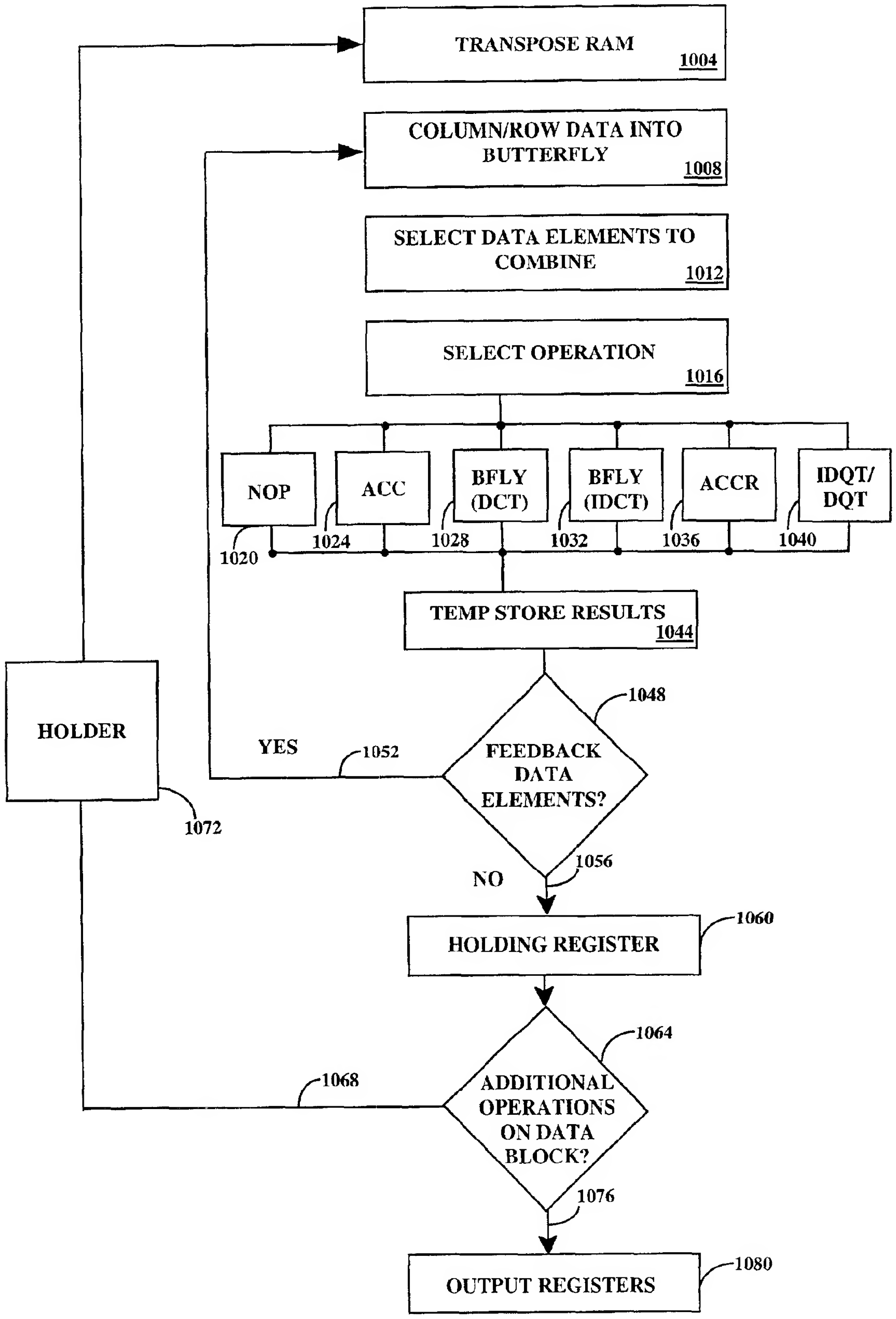

TRANSPOSE RAM
1004
COLUMN/ROW DATA INTO BUTTERFLY
1008
SELECT DATA ELEMENTS TO COMBINE
1012
SELECT OPERATION
1016
NOP
ACC
BFLY (DCT)
BFLY (IDCT)
ACCR
IDQT/ DQT
1020
1024
1028
1032
1036
1040
TEMP STORE RESULTS
1044
FEEDBACK DATA ELEMENTS?
1048
YES
1052
NO
1056
HOLDER
1072
HOLDING REGISTER
1060
ADDITIONAL OPERATIONS ON DATA BLOCK?
1064
1068
1076
OUTPUT REGISTERS
1080

FIG. 10

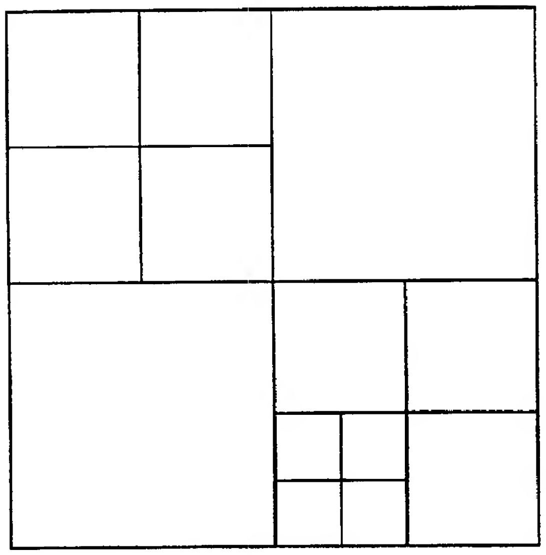


FIG. 11A

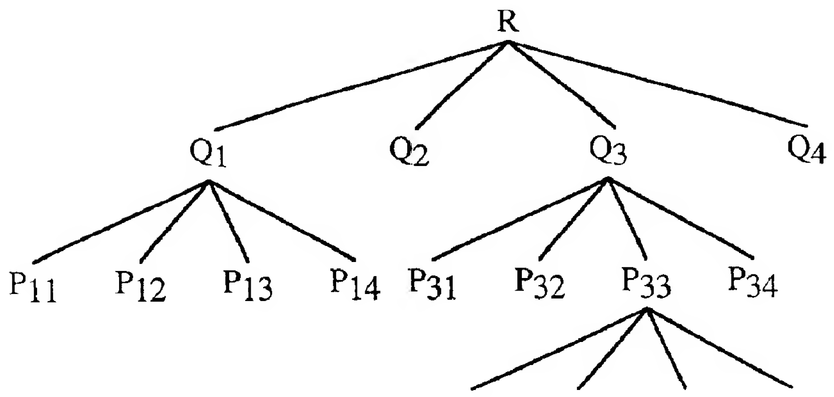


FIG. 11B

1	1	0	1	0	0	0	0	0	0	0	1	0

PQR DATA

FIG. 11C

APPARATUS AND METHOD FOR ENCODING AND COMPUTING A DISCRETE COSINE TRANSFORM USING A BUTTERFLY PROCESSOR

This application claims the benefit of priority of the U.S. Provisional Patent Application Ser. No. 60/291,467, filed May 16, 2001, which is incorporated herein by reference in its entirety.

BACKGROUND OF THE INVENTION

I. Field of the Invention

The present invention relates to digital signal processing. More specifically, the present invention relates to an apparatus and method for determining the transform of a block of encoded data.

II. Description of the Related Art

Digital picture processing has a prominent position in the general discipline of digital signal processing. The importance of human visual perception has encouraged tremendous interest and advances in the art and science of digital picture processing. In the field of transmission and reception of video signals, such as those used for projecting films or movies, various improvements are being made to image compression techniques. Many of the current and proposed video systems make use of digital encoding techniques. Aspects of this field include image coding, image restoration, and image feature selection. Image coding represents the attempts to transmit pictures of digital communication channels in an efficient manner, making use of as few bits as possible to minimize the band width required, while at the same time, maintaining distortions within certain limits. Image restoration represents efforts to recover the true image of the object. The coded image being transmitted over a communication channel may have been distorted by various factors. Source of degradation may have arisen originally in creating the image from the object. Feature selection refers to the selection of certain attributes of the picture. Such attributes may be required in the recognition, classification, and decision in a wider context.

Digital encoding of video, such as that in digital cinema, is an area which benefits from improved image compression techniques. Digital image compression may be generally classified into two categories: loss-less and lossy methods. A loss-less image is recovered without any loss of information. A lossy method involves an irrecoverable loss of some information, depending upon the compression ratio, the quality of the compression algorithm, and the implementation of the algorithm. Generally, lossy compression approaches are considered to obtain the compression ratios desired for a cost-effective digital cinema approach. To achieve digital cinema quality levels, the compression approach should provide a visually loss-less level of performance. As such, although there is a mathematical loss of information as a result of the compression process, the image distortion caused by this loss should be imperceptible to a viewer under normal viewing conditions.

Existing digital image compression technologies have been developed for other applications, namely for television systems. Such technologies have made design compromises appropriate for the intended application, but do not meet the quality requirements needed for cinema presentation.

Digital cinema compression technology should provide the visual quality that a moviegoer has previously experienced. Ideally, the visual quality of digital cinema should attempt to exceed that of a high-quality release print film. At

the same time, the compression technique should have high coding efficiency to be practical. As defined herein, coding efficiency refers to the bit rate needed for the compressed image quality to meet a certain qualitative level. Moreover, the system and coding technique should have built-in flexibility to accommodate different formats and should be cost effective; that is, a small-sized and efficient decoder or encoder process.

One compression technique capable of offering significant levels of compression while preserving the desired level of quality utilizes adaptively sized blocks and sub-blocks of encoded Discrete Cosine Transform (DCT) coefficient data. Although DCT techniques are gaining wide acceptance as a digital compression method, efficient hardware implementation has been difficult.

SUMMARY OF THE INVENTION

The invention provides for efficient hardware implementation of adaptive block sized DCT encoded data. An apparatus to determine a transform of a block of encoded data the block of encoded data comprising a plurality of data elements. An input register is configured to receive a predetermined quantity of data elements. At least one butterfly processor is coupled to the input register and is configured to perform at least one mathematical operation on selected pairs of data elements to produce an output of processed data elements. At least one intermediate register is coupled to the butterfly processor and configured to temporarily store the processed data. A feedback loop is coupled to the intermediate register and the butterfly processor, and where if enabled, is configured to transfer a first portion of processed data elements to the appropriate butterfly processor to perform additional mathematical operations and where if disabled, is configured to transfer a second portion of processed data elements to at least one holding register.

Accordingly, it is an aspect of an embodiment to provide a processor that efficiently implements discrete cosine transform (DCT) and discrete quadtree transform (DQT) techniques.

It is another aspect of an embodiment to provide a processor that efficiently implements inverse discrete cosine transform (IDCT) and inverse discrete quadtree transform (IDQT) techniques.

It is another aspect of an embodiment to implement a processor that is flexible in that the same hardware components may be reconfigured to compute different mathematical operations within the same transform trellis.

It is another aspect of an embodiment to provide an image processor that maintains a high quality image while minimizing image distortion.

It is another aspect of an embodiment to process portions of encoded data in parallel.

It is another aspect of an embodiment to process read, write, and butterfly operations in a single clock cycle.

It is another aspect of an embodiment to provide and implement a control sequencer having the variability to control different block sizes of data and maintain the speed necessary for real-time processing.

It is another aspect of an embodiment to implement a processor such that the processor is configurable to operate on variable block sizes.

BRIEF DESCRIPTION OF THE DRAWINGS

The aspects, features, objects, and advantages of the invention will become more apparent from the detailed description set forth below when taken in conjunction with the drawings in which like reference characters identify correspondingly throughout and wherein:

FIG. **1** is a block diagram of column and row processing of a block of data;

FIG. **2***a* is a block diagram illustrating the flow of data through an encoding process;

FIG. **2***b* is a flow diagram illustrating the flow of data through a decoding process;

FIG. **2***c* is a block diagram illustrating the processing steps involved in variance based block size assignment;

FIG. **3** is a block diagram illustrating an apparatus to compute a transform, such as a discrete cosine transform (DCT) and a discrete quantization transform (DQT), embodying the invention;

FIG. **4** illustrates a DCT trellis that is implemented by the apparatus of FIG. **3**;

FIG. **5** illustrates an IDCT trellis that is implemented by the apparatus of FIG. **3**;

FIG. **6** illustrates a single butterfly processor with input and output multiplexers;

FIG. **7** illustrates a block diagram of a write multiplexer;

FIG. **8** illustrates a block diagram of a butterfly processor;

FIG. **9***a* illustrates a No Operation configuration that may be performed by butterfly processor of FIG. **8**;

FIG. **9***b* illustrates an Accumulate Operation configuration that may be performed by butterfly processor of FIG. **8**;

FIG. **9***c* illustrates a butterfly DCT Operation configuration that may be performed by butterfly processor of FIG. **8**;

FIG. **9***d* illustrates a Butterfly IDCT Operation configuration that may be performed by butterfly processor of FIG. **8**;

FIG. **9***e* illustrates an Accumulate Register Operation configuration that may be performed by butterfly processor of FIG. **8**;

FIG. **9***f* illustrates a DQT/IDQT Operation configuration that may be performed by butterfly processor of FIG. **8**;

FIG. **10** illustrates a flowchart showing the process of calculating a transform, such as a discrete cosine transform (DCT) and a discrete quantization transform (DQT), embodying the invention;

FIG. **11***a* illustrates an exemplary block size assignment;

FIG. **11***b* illustrates the corresponding quad-tree decomposition for the block size assignment of FIG. **11***a;* and

FIG. **11***c* illustrates a corresponding PQR data for the block size assignment of FIG. **11***a*.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

In order to facilitate digital transmission of digital signals and enjoy the corresponding benefits, it is generally necessary to employ some form of signal compression. To achieve high definition in a resulting image, it is also important that the high quality of the image be maintained. Furthermore, computational efficiency is desired for compact hardware implementation, which is important in many applications.

Accordingly, spatial frequency-domain techniques, such as Fourier transforms, wavelet, and discrete cosine transforms (DCT) generally satisfy the above criteria. The DCT has energy packing capabilities and approaches a statistical optimal transform in decorellating a signal. The development of various algorithms for the efficient implementation of DCT further contributes to its mainstream applicability.

The reduction and computational complexity of these algorithms and its recursive structure results in a more simplified hardware scheme. DCTs are generally orthogonal and separable. The fact that DCTs are orthogonal implies that the energy, or information, of a signal is preserved under transformation; that is, mapping into the DCT domain. The fact that DCTs are separable implies that a multidimensional DCT may be implemented by a series of one-dimensional transforms. Accordingly, faster algorithms may be developed for one-dimensional DCTs and be directly extended to multidimensional transforms.

In a DCT, a block of pixels is transformed into a same-size block of coefficients in the frequency domain. Essentially, the transform expresses a block of pixels as a linear combination of orthogonal basis images. The magnitudes of the coefficients express the extent to which the block of pixels and the basis images are similar.

Generally, an image to be processed in the digital domain is composed of pixel data divided into an array of non-overlapping blocks, N×N in size. A two-dimensional DCT may be performed on each block. The two-dimensional DCT is defined by the following relationship:

$$X(k,l)=\frac{\alpha(k)\beta(l)}{N}\sum_{m=0}^{N-1}\sum_{n=0}^{N-1}x(m,n)\cos\left[\frac{(2m+1)\pi k}{2N}\right]\cos\left[\frac{(2m+1)\pi l}{2N}\right],$$

$$0\le k,l\le N-1$$

where

$$\alpha(k),\beta(k)=\begin{cases}1, & \text{if } k=0\\ \sqrt{2}, & \text{if } k\neq 0\end{cases},$$

and

x(m,n) is the pixel location (m,n) within an N×M block, and

X(k,l) is the corresponding DCT coefficient.

Since pixel values are non-negative, the DCT component X(**0**,**0**) is always positive and usually has the most energy. In fact, for typical images, most of the transform energy is concentrated around the component X(**0**,**0**). This energy compaction property makes the DCT technique such an attractive compression method.

It has been observed that most natural images are made up of flat relatively slow varying areas, and busy areas such as object boundaries and high-contrast texture. Contrast adaptive coding schemes take advantage of this factor by assigning more bits to the busy areas and fewer bits to the less busy areas. This technique is disclosed in U.S. Pat. No. 5,021,891, entitled “Adaptive Block Size Image Compression Method and System,” assigned to the assignee of the present invention and incorporated herein by reference. DCT techniques are also disclosed in U.S. Pat. No. 5,107,345, entitled “Adaptive Block Size Image Compression Method And System,” assigned to the assignee of the present invention and incorporated herein by reference. Further, the use of the ABSDCT technique in combination with a Differential Quadtree Transform technique is discussed in U.S. Pat. No. 5,452,104, entitled “Adaptive Block Size Image Compression Method And System,” also assigned to the assignee of the present invention and incorporated herein by reference. The systems disclosed in these patents utilizes what is referred to as “intra-frame” encoding, where each frame of image data is encoded without regard to the content of any other frame. Using the ABSDCT technique, the achievable data rate may be greatly reduced without discernible degradation of the image quality.

Using ABSDCT, a video signal will generally be segmented into frames and blocks of pixels for processing. The DCT operator is one method of converting a time-sampled signal to a frequency representation of the same signal. By converting to a frequency representation, DCT techniques have been shown to allow for very high levels of compression, as quantizers can be designed to take advantage of the frequency distribution characteristics of an image. In a preferred embodiment, one 16×16 DCT is applied to a first ordering, four 8×8 DCTs are applied to a second ordering, 16 4×4 DCTs are applied to a third ordering, and 64 2×2 DCTs are applied to a fourth ordering.

For image processing purposes, the DCT operation is performed on pixel data that is divided into an array of non-overlapping blocks. Note that although block sizes are discussed herein as being N×N in size, it is envisioned that various block sizes may be used. For example, an N×M block size may be utilized where both N and M are integers with M being either greater than or less than N. Another important aspect is that the block is divisible into at least one level of sub-blocks, such as N/i×N/i, N/i×N/j, N/i×M/j, and etc. where i and j are integers. Furthermore, the exemplary block size as discussed herein is a 16×16 pixel block with corresponding block and sub-blocks of DCT coefficients. It is further envisioned that various other integers such as both even or odd integer values may be used, e.g., 9×9.

A color signal may be converted from RGB space to YC_1C_2 space, with Y being the luminance, or brightness, component, and C_1 and C_2 being the chrominance, or color, components. Because of the low spatial sensitivity of the eye to color, many systems sub-sample the C_1 and C_2 components by a factor of four in the horizontal and vertical directions. However, the sub-sampling is not necessary. A full resolution image, known as 4:4:4 format, may be either very useful or necessary in some applications such as those referred to as covering digital cinema. Two possible YC_1C_2 representations are, the YIQ representation and the YUV representation, both of which are well known in the art. It is also possible to employ a variation of the YUV representation known as YCbCr.

FIGS. **1***a* and **1***b* illustrate column and row processing of a N×N block of encoded data **100** and **120**. An N dimensional transform may be performed as a cascade of N one-dimensional transforms. For example, a 2×2 DCT is performed as a cascade of two one-dimensional DCT processes, first operating on each column and then operating on each row. A first column m (**104**) is processed, followed by column m+1 (**108**), followed by column m+2 (**112**), and so on through column n (**116**). After the columns are processed, the rows **120** are processed as illustrated in FIG **1***b*. First, row m (**124**) is processed, followed by row m+1 (**128**), row m+2 (**132**) and so on through row n (**136**).

Similarly, another example may be an 8×8 block of data needing IDCT processing. The 8×8 block may be broken into four two-dimensional IDCTs. Each two-dimensional IDCT may then be processed in the same manner with respect to the two-dimensional DCT described with respect to FIGS. **1***a* and **1***b*.

FIG. **2***a* illustrates a block diagram **250** of the flow of encoded data during an encoding process. In the encoding process, encoded data is transformed from the pixel domain to the frequency domain. FIG. **2***b* illustrates a block diagram **254** of the flow of encoded data through a decoding process. In the decoding process, encoded data is transformed from the frequency domain to the pixel domain. As illustrated in

the encode process **250**, a block sized assignment (BSA) of the encoded data is first performed (**258**). In an aspect of an embodiment, each of the Y, Cb, and Cr components is processed without sub-sampling. Thus, an input of a 16×16 block of pixels is provided to the block size assignment element **258**, which performs block size assignment in preparation for video compression.

The block size assignment element **258** determines the block decomposition of a block based on the perceptual characteristics of the image in the block. Block size assignment subdivides each 16×16 block into smaller blocks in a quad-tree fashion depending on the activity within a 16×16 block. The block size assignment element **258** generates a quad-tree data, called the PQR data, whose length can be between 1 and 21 bits. Thus, if block size assignment determines that a 16×16 block is to be divided, the R bit of the PQR data is set and is followed by four additional bits of Q data corresponding to the four divided 8×8 blocks. If block size assignment determines that any of the 8×8 blocks is to be subdivided, then four additional bits of P data for each 8×8 block subdivided are added.

Data is divided into block sizes, such as 2×2, 4×4, 8×8, and 16×16. An encode data processor then performs a transform (DCT/DQT) of the encoded data (**262**), as is described with respect to FIG. **3**. After the DCT/DQT process **262** is completed, a quantization process (QB) **266** is performed on the encoded data. This completes transformation of encoded data from the pixel domain to the frequency domain.

In an embodiment, the DCT coefficients are quantized using frequency weighting masks (FWMs) and a quantization scale factor. A FWM is a table of frequency weights of the same dimensions as the block of input DCT coefficients. The frequency weights apply different weights to the different DCT coefficients. The weights are designed to emphasize the input samples having frequency content that the human visual system is more sensitive to, and to de-emphasize samples having frequency content that the visual system is less sensitive to. The weights may also be designed based on factors such as viewing distances, etc.

Huffman codes are designed from either the measured or theoretical statistics of an image. It has been observed that most natural images are made up of blank or relatively slowly varying areas, and busy areas such as object boundaries and high-contrast texture. Huffman coders with frequency-domain transforms such as the DCT exploit these features by assigning more bits to the busy areas and fewer bits to the blank areas. In general, Huffman coders make use of look-up tables to code the run-length and the non-zero values.

The weights are selected based on empirical data. A method for designing the weighting masks for 8×8 DCT coefficients is disclosed in ISO/IEC JTC1 CD 10918, "Digital compression and encoding of continuous-tone still images—part 1: Requirements and guidelines," International Standards Organization, 1994, which is herein incorporated by reference. In general, two FWMs are designed, one for the luminance component and one for the chrominance components. The FWM tables for block sizes 2×2, 4×4 are obtained by decimation and 16×16 by interpolation of that for the 8×8 block. The scale factor controls the quality and bit rate of the quantized coefficients.

Thus, each DCT coefficient is quantized according to the relationship:

$$DCT_q(i,j) = \left\lfloor \frac{8 * DCT(i,j)}{fwm(i,j) * q} \pm \frac{1}{2} \right\rfloor$$

where DCT(i,j) is the input DCT coefficient, fwm(i,j) is the frequency weighting mask, q is the scale factor, and DCTq (i,j) is the quantized coefficient. Note that depending on the sign of the DCT coefficient, the first term inside the braces is rounded up or down. The DQT coefficients are also quantized using a suitable weighting mask. However, multiple tables or masks can be used, and applied to each of the Y, Cb, and Cr components.

The quantized coefficients are provided to a zigzag scan serializer **268**. The serializer **268** scans the blocks of quantized coefficients in a zigzag fashion to produce a serialized stream of quantized coefficients. A number of different zigzag scanning patterns, as well as patterns other than zigzag may also be chosen. A preferred technique employs 8×8 block sizes for the zigzag scanning, although other sizes, such as 4×4 or 16×16, may be employed.

Note that the zigzag scan serializer **268** may be placed either before or after the quantizer **266**. The net results are the same.

In any case, the stream of quantized coefficients is provided to a variable length coder **269**. The variable length coder **269** may make use of run-length encoding of zeros followed by encoding. This technique is discussed in detail in aforementioned U.S. Pat. Nos. 5,021,891, 5,107,345 and 5,452,104, and in pending U.S. patent application Ser. No. <000163>, which is incorporated by reference and is summarized herein. A run-length coder takes the quantized coefficients and notes the run of successive coefficients from the non-successive coefficients. The successive values are referred to as run-length values, and are encoded. The non-successive values are separately encoded. In an embodiment, the successive coefficients are zero values, and the non-successive coefficients are non-zero values. Typically, the run length is from 0 to 63 bits, and the size is an AC value from 1–10. An end of file code adds an additional code—thus, there is a total of 641 possible codes.

In the decoding process, encoded data in the frequency domain is converted back into the pixel domain. A variable length decoder **270** produces a run-length and size of the data and provides the data to an inverse zigzag scan serializer **271** that orders the coefficients according to the scan scheme employed. The inverse zigzag scan serializer **271** receives the PQR data to assist in proper ordering of the coefficients into a composite coefficient block. The composite block is provided to an inverse quantizer **272**, for undoing the processing due to the use of the frequency weighting masks.

A finger printer (H2O) **273** is then performed on the encoded data. The finger printer places a watermark or other identifier information on the data. The watermark may be recovered at a later time, to reveal identifier information. Identifier information may include information such as where and when material was played, and who was authorized to play such material. Following the finger printer **273**, a decoder data process **274** (IDQT/IDCT) is commenced, which is described in detail with respect to FIG. **4**. After the data is decoded, the data is sent to the Frame Buffer Interface (FBI) **278**. The FBI is configured to read and write uncompressed data a frame at a time. In an embodiment, the FBI has a capacity of four frames, although it is contemplated that the storage capacity may be varied.

Referring now to FIG. **2***c*, a flow diagram showing details of the operation of the block size assignment element **258** is provided. The algorithm uses the variance of a block as a metric in the decision to subdivide a block. Beginning at step **202**, a 16×16 block of pixels is read. At step **204**, the variance, v**16**, of the 16×16 block is computed. The variance is computed as follows:

$$\text{var} = \frac{1}{N^2}\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} x_{i,j}^2 - \left(\frac{1}{N^2}\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} x_{i,j}\right)^2$$

where N=16, and $x_{i,j}$ is the pixel in the ith row, jth column within the N×N block. At step **206**, first the variance threshold T16 is modified to provide a new threshold T'**16** if the mean value of the block is between two predetermined values, then the block variance is compared against the new threshold, T'**16**.

If the variance v**16** is not greater than the threshold **T16**, then at step **208**, the starting address of the 16×16 block is written, and the R bit of the PQR data is set to 0 to indicate that the 16×16 block is not subdivided. The algorithm then reads the next 16×16 block of pixels. If the variance v**16** is greater than the threshold **T16**, then at step **210**, the R bit of the PQR data is set to 1 to indicate that the 16×16 block is to be subdivided into four 8×8 blocks.

The four 8×8 blocks, i=1:4, are considered sequentially for further subdivision, as shown in step **212**. For each 8×8 block, the variance, v**8**$_i$, is computed, at step **214**. At step **216**, first the variance threshold T8 is modified to provide a new threshold **T8** if the mean value of the block is between two predetermined values, then the block variance is compared to this new threshold.

If the variance v**8**$_i$ is not greater than the threshold **T8**, then at step **218**, the starting address of the 8×8 block is written, and the corresponding Q bit, Q_i, is set to 0. The next 8×8 block is then processed. If the variance v**8**$_j$ is greater than the threshold T8, then at step **220**, the corresponding Q bit, Q_i, is set to 1 to indicate that the 8×8 block is to be subdivided into four 4×4 blocks.

The four 4×4 blocks, j_i=1:4, are considered sequentially for further subdivision, as shown in step **222**. For each 4×4 block, the variance, v**4**$_{ij}$, is computed, at step **224**. At step **226**, first the variance threshold **T4** is modified to provide a new threshold T'**4** if the mean value of the block is between two predetermined values, then the block variance is compared to this new threshold.

If the variance v**4**$_{ij}$ is not greater than the threshold T4, then at step **228**, the address of the 4×4 block is written, and the corresponding P bit, P_{ij}, is set to 0. The next 4×4 block is then processed. If the variance v**4**$_{ij}$ is greater than the threshold T4, then at step **230**, the corresponding P bit, P_{ij}, is set to 1 to indicate that the 4×4 block is to be subdivided into four 2×2 blocks. In addition, the address of the 4 2×2 blocks is written.

The thresholds **T16**, **T8**, and T4 may be predetermined constants. This is known as the hard decision. Alternatively, an adaptive or soft decision may be implemented. The soft decision varies the thresholds for the variances depending on the mean pixel value of the 2N×2N blocks, where N can be 8, 4, or 2. Thus, functions of the mean pixel values, may be used as the thresholds.

For purposes of illustration, consider the following example. Let the predetermined variance thresholds for the Y component be 50, 1100, and 880 for the 16×16, 8×8, and 4×4 blocks, respectively. In other words, **T16**=50, **T8**=1100, and **T16**=880. Let the range of mean values be 80 and 100. Suppose the computed variance for the 16×16 block is 60. Since 60 and its mean value 90 are greater than **T16**, the

16×16 block is subdivided into four 8×8 sub-blocks. Suppose the computed variances for the 8×8 blocks are 1180, 935, 980, and 1210. Since two of the 8×8 blocks have variances that exceed **T8**, these two blocks are further subdivided to produce a total of eight 4×4 sub-blocks. Finally, suppose the variances of the eight 4×4 blocks are 620, 630, 670, 610, 590, 525, 930, and 690, with the first four corresponding means values 90, 120, 110, 115. Since the mean value of the first 4×4 block falls in the range (80, 100), its threshold will be lowered to T'4=200 which is less than 880. So, this 4×4 block will be subdivided as well as the seventh 4×4 block. The resulting block size assignment is illustrated in FIG. **11***a*. The corresponding quad-tree decomposition is illustrated in FIG. **11***b*. The PQR data generated by this block size assignment is illustrated in FIG. **11***c*.

Note that a similar procedure is used to assign block sizes for the color components C_1 and C_2. The color components may be decimated horizontally, vertically, or both. Additionally, note that although block size assignment has been described as a top down approach, in which the largest block (16×16 in the present example) is evaluated first, a bottom up approach may instead be used. The bottom up approach will evaluate the smallest blocks (2×2 in the present example) first.

The PQR data, along with the addresses of the selected blocks, are provided to a DCT/DQT element **262**. The DCT/DQT element **262** uses the PQR data to perform discrete cosine transforms of the appropriate sizes on the selected blocks. Only the selected blocks need to undergo DCT processing. The DQT is also used for reducing the redundancy among the DC coefficients of the DCTs. A DC coefficient is encountered at the top left comer of each DCT block. The DC coefficients are, in general, large compared to the AC coefficients. The discrepancy in sizes makes it difficult to design an efficient variable length coder. Accordingly, it is advantageous to reduce the redundancy among the DC coefficients. The DQT element performs 2-D DCTs on the DC coefficients, taken 2×2 at a time. Starting with 2×2 blocks within 4×4 blocks, a 2-D DCT is performed on the four DC coefficients. This 2×2 DCT is called the differential quad-tree transform, or DQT, of the four DC coefficients. Next, the DC coefficient of the DQT along with the three neighboring DC coefficients with an 8×8 block are used to compute the next level DQT. Finally, the DC coefficients of the four 8×8 blocks within a 16×16 block are used to compute the DQT. Thus, in a 16×16 block, there is one true DC coefficient and the rest are AC coefficients corresponding to the DCT and DQT.

Within a frame, each 16×16 block is computed independently. Accordingly, the processing algorithm used for a given block may be changed as necessary, as determined by the PQR.

FIG. **3** is a block diagram illustrating computation of the DCT/DQT and the IDQT/IDCT of a block of encoded data **300**. In encode mode, as illustrated in FIG. **3**, the encoded data is initially in the pixel domain. As the encoded data is processed through intermediate steps, the encoded data is transformed into the frequency domain. In decode mode, the encoded data is initially in the frequency domain. As the encoded data is processed through intermediate steps, the encoded data is transformed into the pixel domain.

Referring to FIG. **3**, at least one M×N block of encoded data is stored in a transpose RAM **304**. The transpose RAM **304** may contain one or more blocks of M×N data. In an embodiment with two blocks of encoded data, one is configured to contain a current M×N block of data **308**, and the other configure to contain a next block of M×N data **312**.

The blocks of data **308** and **312** are transferred to transpose RAM **304** from the block size assignment **208** as illustrated in FIG. **2***a* (in encode mode) or the fingerprinter **220** as illustrated in FIG. **2***b* (in decode mode). In an embodiment, the transpose RAM **304** may be a dual port RAM, such that a transpose RAM interface **316** processes the current block of data **308** and receives the next block of data from the fingerprinter **220**. The transpose RAM interface **316** controls timing and may have buffered memory to allow blocks of data to be read from and written to the transpose RAM **304**. In an embodiment, the transpose RAM **304** and transpose RAM interface **316** may be responsive to one or more control signals from a control sequencer **324**.

Encoded data enters a data processor **328** from transpose RAM **304** (or through the transpose RAM interface **316**) into one or more input registers **332**. In an embodiment, there are 16 input registers **332**. In an embodiment, the data processor **328** first processes column data, followed by row data, as illustrated in FIG. **1**. The data processor **328** may alternatively process the rows followed by the columns, however, the following description assumes that column data is processed prior to row data. The input register **332** comprises of a single column encoded data of the 16×16 block. The data processor **328** computes the transform by performing mathematical operations on the encoded data, column by column, and writes the data back into the transpose RAM **304**. After the columns of data are processed, the data processor **328** processes each row of encoded data. After each row of encoded data is processed, the data processor **328** outputs the data through an output register **352**.

In an embodiment, the block of data is a 16×16 block of encoded data, although it is contemplated that any size block of data may be used, such as 32×32, 8×8, 4×4, or 2×2, or combinations thereof. Accordingly, as the data processor **328** is processing a block of data from the transpose RAM **304** (for example, the current M×N block of data **308**), the transpose RAM interface **316** receives the next block of data **312** from the BSA **208** (encode mode) or the fingerprinter **220** (decode mode). When the data processor **328** has completed processing of the current block of data **308**, the transpose RAM interface **316** reads the next block of data **312** from the transpose RAM **304** interface and loads it into data processor **328**. As such, data from the transpose RAM **304** toggles between the current block of data **308** and the next block of data **312** as dictated by the transpose RAM interface **316** and the control sequencer **324**.

The data processor **328** comprises input register **332**, at least one butterfly processor within a monarch butterfly cluster **336** and at least one intermediate data register **340**. Data processor **328** may also comprise a holding register **344**, a write mutliplexer **348**, and output data register **352**. Monarch butterfly cluster **336** may further comprise a first input multiplexer **356**, and intermediate data register **340** further comprises a second input multiplexer **360**. The aforementioned components of data processor **328** are preferably controlled by the control sequencer **324**.

In operation, for a given column or row of data, the input register **332** is configured to receive the encoded data through the transpose RAM interface **316** from the transpose RAM **304**. The control sequencer **324** enables certain addresses of the input register to send the data through input multiplexer **356**. The data input is resequenced as by selection through input multiplexer **356** such that the proper pairs of encoded data are selected for mathematical operations. Controlled by the control sequencer **324**, the input multiplexer **356** passes the data to the monarch butterfly cluster

336. The monarch butterfly cluster **336** comprises one or more butterfly processors. In an embodiment, the monarch butterfly cluster **336** comprises four individual butterfly processors **364, 368, 372**, and **376**, and control sequencer **324** routes encoded data through input multiplexer **356** to the appropriate butterfly processor.

Each individual butterfly processor **364, 368, 372** or **376** is capable of performing one-dimensional transforms, such as the DCT, IDCT, DQT and IDQT. A one-dimensional transform typically involve arithmetic operations, such as simple adders, subtractors, or a multiplier. After a portion of a one-dimensional transform is performed on a pair of data elements, the resulting output is transferred to the intermediate data register **340**. Intermediate data register **340** may be responsive to the control sequencer **324**. The control sequencer may be a device such as a state machine, a microcontroller, or a programmable processor. In an embodiment in which the intermediate data register **340** is responsive to the control sequencer **324**, selected data elements stored in the intermediate data register **340** are fed back to appropriate butterfly processor using a feedback path **380** and through first input multiplexer **356**, to be processed again (i.e., another portion of a one-dimensional transform). This feedback loop continues until all one-dimensional processing for the encoded data is completed. When the processing of the data is completed, the data from the intermediate data register **340** is written to the WRBR holding register **344**. If the data being processed is column data, the data is written from the WRBR holding register **344** through the write multiplexer **348** and stored back into the transpose RAM **304**, so that row processing may begin. The write multiplexer **348** is controlled to resequence the processed column data back into its original sequence. If the holding register data is row data (and thus, all of the column processing is complete), the data is routed to the output register **352**. The control sequencer **324** may then control output of data from the daisy chain multiplexer and output data register **352**.

FIG. **4** illustrates a DCT trellis that may be implemented in encode mode by the data path processor **328** of FIG. **3**. Similarly, FIG. **5** illustrates an IDCT trellis that may be implemented in decode mode by the data path processor **328** of FIG. **3**. As dictated by the PQR data and/or depending on the type of computation being performed, the control sequencer **324** may select different pairs of elements of encoded data to combine and performs portions of a one-dimensional transform. For example, in the trellis of FIG. **4**, eight operations occur in column **404**. The operations illustrated are as follows: x(**0**)+x(**7**), x(**1**)+x(**6**), x(**3**)+x(**4**), x(**2**)+x(**5**), x(**0**)−x(**7**), x(**1**)−x(**6**), x(**3**)−x(**4**) and x(**2**)−x(**5**). Each of the butterfly processors **364, 368, 372** and **376** (as shown FIG. **3**) handles one of the four operations in a given clock cycle. Thus, for example, butterfly processor **364** computes the operation of x(**0**)+x(**7**) and x(**0**)−x(**7**), butterfly processor **368** computes the operation of x(**1**)+x(**6**) and x(**1**)−x(**6**), butterfly processor **372** computes the operation of x(**3**)+x(**4**) and x(**3**)−x(**4**), and butterfly processor **376** computes the operation of x(**2**)+x(**5**) and x(**2**)−x(**5**), all in the same clock cycle. The results of each of these operations may be temporarily stored in a pipeline register or in the intermediate data register **340**, and then routed to the input multiplexer **360**. Operation of the pipeline register is described in the specification with respect to FIG. **9***c* and **9***d*.

Optionally, in the next clock cycle, the remaining four multiplication operations are computed using the same four butterfly processors. Accordingly, butterfly processor **364** computes [x(**0**)−x(**7**)]*(½C^1_{16}), butterfly processor **368** computes [x(**1**)−x(**6**)]* (½C^3_{16}), butterfly processor **372** computes [x(**3**)−x(**4**)]*(½C^7_{16}) and butterfly processor **376** computes [x(**2**)−x(**5**)]*(½C^5_{16}). The results of these computations are temporarily stored in the intermediate data register **340**. As computations are completed, the encoded data is not in the same sequence that the encoded data was in when originally input. Accordingly, control sequencer **324** and input multiplexer **356** resequences encoded data, or partially processed encoded data after each feed back loop, as necessary.

In the following clock cycle, computations are processed for column **408**, the results of which are again stored in the intermediate data register **340** are fed back through input multiplexer **360**. Again, the fed back encoded data, now partially processed, is resequenced such that the right portions of encoded data are routed to the appropriate butterfly processor. Accordingly, butterfly processor **364** processes b(**0**)+b(**2**) and b(**0**)−b(**2**). Similarly, butterfly processor **368** computes b(**1**)+b(**3**) and b(**1**)−b(**3**), butterfly processor **372** computes b(**4**)+b(**6**) and b(**4**)−b(**6**)and butterfly processor **376** computes b(**5**)+b(**7**) and b(**5**)−b(**7**). The resulting computations are again stored with the intermediate data register **340** or a pipeline register, and routed through the input multiplexer **360**. In the next clock cycle, multiplications are performed by ½ C^1_8, ½C^3_8, ½C^1_8, and ½C^3_8, in the same manner as described with respect to column **404**. Thus, butterfly processor **364** computes b(**0**)−b(**2**)*½ C^1_8, butterfly processor **368** computes b(**1**)−b(**3**)*½ C^3_8, butterfly processor **372** computes b(**4**)−b(**6**)*½ C^1_8, butterfly processor **376** computes b(**5**)−b(**7**)*½ C^3_8.

In the next clock cycle, computations are processed for column **412** for values in the d(**0**) through d(**7**) positions are computed, the results of which are again stored in the intermediate data register **340** and are fed back into input multiplexer **360**. Accordingly, each butterfly processor computes each stage of each input, such that butterfly processor **364** computes the operation of d(**0**)+d(**1**) and d(**0**)−d(**1**), butterfly processor **368** computes the operation of d(**2**)+d(**3**) and d(**2**)−d(**3**), butterfly processor **372** computes the operation of d(**4**)+d(**5**) and d(**4**)−d(**5**), and butterfly processor **376** computes the operation of d(**6**)+d(**7**) and d(**6**)−d(**7**), all in the same clock cycle. In the following clock cycle, multiplications by ½ C^1_4 are computed in the same manner as described with respect to columns **404** and **408**.

Column **416** illustrates the next set of mathematical operations computed by the butterfly processors in the next clock cycle. As shown in the example of FIG. **4** in column **416**, only two operations are needed during this clock cycle: namely, the sum of the f(**2**) and f(**3**) components, and the sum of the f(**6**) and f(**7**) components. Accordingly, butterfly processor **364** computes f(**2**)+f(**3**), and butterfly processor **368** computes f(**6**)+f(**7**).

In the following clock cycle, the computations expressed in column **420** are processed. As such, values for h(**4**), h(**5**) and h(**6**) are computed. Accordingly, butterfly processor **364** computes h(**4**)+h(**6**), butterfly processor **368** computes h(**5**)+h(**8**), and butterfly processor **372** computes h(**5**)+h(**6**).

As readily observable, FIG. **5** illustrates an IDCT trellis that operates in a similar manner, but an opposite sequence than the trellis described with respect to FIG. **4**. The IDCT trellis is utilized in the decode process, as opposed to the DCT trellis which operates in the encode process. The butterfly processors **364, 368, 372** and **376** operate in the same manner as described with respect to FIG. **4**, taking advantage of efficiencies in parallel processing. Both in the encode and decode process, a significant advantage of an embodiment is the reuse of the same hardware for each stage of the trellis. Accordingly, the hardware is used for the computations illustrated in column **504** is the same as the hardware used for computations of columns **508**, **512**, **516** and **520**. Similarly, the hardware used for the computations illustrated in column **404** is the same as the hardware used for computations of columns **408**, **412**, **416** and **420**.

Once the final results representing the end of the trellis in FIG. **4** are computed, the data is transferred from the intermediate data register **340** to the holding register **344**. The holding register **344** and output data register **352** are controlled by control sequencer **324**. If data is column data, the data is transferred to the write multiplexer **348** and stored back into the transpose RAM **304**. Again, the encoded data is resequenced to reflect the original sequence of the encoded data. If the data is row data, all computations are therefore completed, and the data is transferred from the holding register **344** to the output data register **352**.

FIG. **6** illustrates an example of a single butterfly processor with one or more input and output multiplexers **600**. In an embodiment, data output from one or more intermediate data registers **340** (see FIG. **3**) are coupled to an input portal of input multiplexer **604**. In an embodiment, the data output from each of the intermediate data registers **340** is input into the butterfly processor to a first multiplexer **608** and a second multiplexer **612**. Data output from the input AR register **332** (see FIG. **3**) is also transferred through the input multiplexer **604**. Specifically, the output of AR register AR(**0**) and AR(**8**) are coupled to the input of multiplexer **616**, and the outputs of AR(**1**), AR(**8**), AR(**9**) and AR(**15**) are coupled to the input of multiplexer **620**. Multiplexers **624** and **628** select either the signal coming from the AR or the BR register as dictated by the control sequencer **324** (illustrated in FIG. **3**). Accordingly, multiplexer **624** selects either the data from multiplexer **608** or **616**, and multiplexer **628** selects either the data from multiplexer **620** or multiplexer **612**. The outputs of the multiplexers **624** and **628** are thus coupled to the input of the individual butterfly processor **632**. Butterfly processor **632** computes a stage of the DCT/IDCT/DQT/IDQT transform, as described with respect to FIGS. **3**, **4** and **5**. The two outputs of the butterfly processor **632**, outputs **636** and **638**, are each coupled to the input of each intermediate data multiplexers **642** and **646**. Data is then selected from the multiplexers **642** and **646** to a bank of intermediate registers **650**. In an embodiment, there are sixteen such intermediate multiplexers and data registers.

FIG. **7** illustrates a block diagram of a write multiplexer. As illustrated in FIG. **3**, the even outputs of the intermediate data register **340** are input into a multiplexer **704**, and the odd outputs of the intermediate data register **340** are input into a multiplexer **708**. The data in each of the intermediate registers are resequenced by multiplexers **704**, **708**, **712** and **716** as controlled by the control sequencer **324** illustrated in FIG. **3**, and stored in 17-bit registers **720** and **724**, respectively. The resequenced data is then stored in the transpose RAM **304**.

FIG. **8** illustrates operation of each butterfly processor **800**. In an embodiment, four butterfly processors are implemented. However, it is contemplated that any number of butterfly processors may be implemented, subject to timing and size constraints. Data enters the butterfly through inputs **804** and **808**. In an embodiment, input **804** sometimes represents the DC value, and passes through a truncator **812**. The truncator **812** is responsible for the 1/N function, as described with respect to the two-dimensional DCT equation infra. The DC value of input **804** is seventeen bits—a single sign bit plus sixteen integer bits. The truncator **812** truncates n bits from the DC value input data to create a truncated DC value **816**, where n is four bits if the data being processed is a 16×16 block, n is three bits if the data being processed is a 8×8 block, n is two bits if the data being processed is a 4×4 block, and n is one bit if the data being processed is a 2×2 block. If the input is an AC value, truncator **812** is bypassed and routed to a first selector **814**. First selector **814** then selects either the truncated DC value **816** or the AC value from input A **804**. In this embodiment, no fractional bits are used, although it is contemplated that fractional bits may be used.

The output of first selector **816** is routed to a delay **820** and a second selector **824**. When the output of selector value **816** is routed to delay **820**, the truncated DC value is may be held for a clock cycle before being routed to second selector **824**. In an embodiment, delay **820** is a register. Selection of data in second selector **824** is a function of the type of mathematical operation that is to be performed on the data. A control word **826**, preferably routed from the control sequencer, triggers second selector **824**. As illustrated throughout FIG. **8**, control word **826** provides control for a number of components. Again depending upon the type of mathematical operation to be performed, the data then passes to an adder **832** or a subtractor **836**. A third selector **828** also receives the delayed output value from the delay **820**, along with input **808**. Again, selection of data in third selector **828** is a function of the type of mathematical operation that is to be performed on the data.

As the data is either added or subtracted, the data is then passed to either a fourth selector **840** or a fifth selector **844** for output from the butterfly processor **800**. Input **804** is also passed to fourth selector **840**, and input **808** is passed to fifth selector **844**. In encode mode, the data may also be routed to sixth selector **848**. In an embodiment, in encode mode, data is routed through an encode delay **852** before being routed to the sixth selector **848**.

The second input, input **808**, passes through the third selector **828** and the sixth selector **848**. If input **808** is selected by sixth selector **848**, the data is routed to a multiplier **856**, where input **808** is multiplied by a scalar **860**. The multiplication process with scalar **860** scales the data to produce a scaled output **864**. In an embodiment, the scalar **860** is selected based on B. G. Lee's algorithm. In an embodiment, the scaled output **864** is then routed to a formatter **868**. The formatter **868** rounds and saturates the data from a twenty-four bit format, a sign bit, sixteen integer bits and seven fractional bit, to a seventeen bit format. Thus, the formatted scaled output **872** is seventeen bits as opposed to twenty bits in length. Treatment of the data in this manner allows precision to be maintained when making calculations, but using fewer bits to represent the same data, which in turn saves hardware space. The formatted scaled output **872** is routed through a delay **876** to third selector **828** and fifth selector **844**, for further processing.

FIGS. **9***a*–**9***f* illustrate various mathematical operations capable of being performed by each butterfly processor. FIG. **9***a* illustrates a NO operation that may be performed by the butterfly processor **900**. Given two inputs, input A (**902**) and input B (**904**), each input is simply passed through to output C (**906**) and output D (**908**). Accordingly, in a NO operation, C=A and D=B.

FIG. **9***b* illustrates an accumulate operation performed by the butterfly processor **910**. Given two inputs, input A (**912**) and input B (**914**), output C (**916**) represents the sum of A+B. Input A (**912**) and input B (**914**) are combined by an adder **913**. Output D (**918**) represents a pass through of input B (**914**). Accordingly, in an accumulate operation, C=A+B and D=B.

FIG. 9*c* illustrates a butterfly DCT operation performed by the butterfly processor **920**. Given two inputs, input A (**922**) and input B (**924**), output C (**926**) represents the sum of input A (**922**) and input B (**924**), such that C=A+B. Input **922** and input **924** are combined by an adder **923**. Output D (**928**) represents a subtracter of input A (**922**) and B (**924**) and multiplied by coefficient CF (**930**), such that the D=CF×(A−B). Input **924** is subtracted from input **922** by a subtractor **925**, and then multiplied by a multiplier **927**. Optionally, pipeline registers **932** and **934** may be used to temporarily store the intermediate product until the next clock cycle.

FIG. 9*d* illustrates a butterfly IDCT operation performed by the butterfly processor **936**. Given two inputs, input A (**938**) and input B (**940**), the output C (**942**) represents the sum of input A (**938**) and input B (**940**) multiplied by a coefficient CF (**943**), such that the output C=A+(B×CF). Input B (**940**) is multiplied by coefficient CF (**943**) by multiplier **945**, and then added to input A (**938**) by adder **947**. Similarly, output D (**944**) represents the difference of input A (**938**) and input B (**940**) multiplied by a coefficient CF (**943**), such that D=A−(B×CF). Input B (**940**) is multiplied by coefficient CF (**943**) by multiplier **945**, and then subtracted from input A (**938**) by subtractor **949**. Optionally, pipeline registers **946** and **948** may store intermediate products to be computed in the next clock cycle.

FIG. 9*e* illustrates an accumulate register operation performed by the butterfly processor **950**. Given two inputs, input A (**952**) and input AREG (**954**), output C (**956**) represents the sum of input A and AREG such that C=A+AREG. As opposed to an input value, AREG may also be a value stored from a previous clock cycle in a register **951**. Input A (**952**) is added to AREG (**954**) by adder **953**.

FIG. 9*f* represents a DQT/IDQT operation performed by the butterfly processor **958**. Given two inputs, input A (**960**) and input B (**962**), output C (**964**) represents the sum of inputs A and B, such that C=A+B. Similarly, output D (**966**) represents the difference of inputs A and B, such that D=A−B. Input A (**960**) and input B (**962**) are combined by an adder **963**. Input B (**962**) is subtracted from input A (**960**) by a subtractor **965**.

The process of calculating a transform of image data **1000** is illustrated in FIG. **10**, and may be implemented in a structure as described with respect to FIG. **3**. The process is easily configured for frequency domain techniques such as the DCT, IDCT, DQT and IDQT. A column or row of data initially resides in a transpose RAM **1004** and is transferred into a holding register **1008** in the butterfly processor. Individual data elements of the block of data are selected to be combined **1012**, and a mathematical operation to be performed on the individual data elements is selected **1016**. Mathematical operations that may be performed are described with respect to FIG. 9*a*–9*f*, and include no operation **1020**, an accumulate **1024**, a DCT butterfly **1028**, an IDCT butterfly **1032**, an accumulate register **1036** and a DQT/IDQT butterfly **1040**. The results of the mathematical operation are temporarily stored **1044**. A feedback decision **1048** is then made based on whether further mathematical operations are needed. In an embodiment, the feedback decision is controlled by the control sequencer, as described with respect to FIG. **3**. If the data is fed back **1052**, the data is fed back to the holding register **1008**, and the process is repeated. If the data is not fed back **1056**, the data is transferred to an output holding register **1060**. Another decision **1064** is made as to whether additional mathematical operations are needed for the column or row of data. If so (**1068**), the column or row of data is transferred to a holder **1072** and then written back into the transpose RAM **1004**. If not (**1076**), the block of data is transferred to output data registers **1080**.

As examples, the various illustrative logical blocks, flowcharts, and steps described in connection with the embodiments disclosed herein may be implemented or performed in hardware or software with an application-specific integrated circuit (ASIC), a programmable logic device, discrete gate or transistor logic, discrete hardware components, such as, e.g., registers and FIFO, a processor executing a set of firmware instructions, any conventional programmable software and a processor, or any combination thereof. The processor may advantageously be a microprocessor, but in the alternative, the processor may be any conventional processor, controller, microcontroller, or state machine. The software could reside in RAM memory, flash memory, ROM memory, registers, hard disk, a removable disk, a CD-ROM, a DVD-ROM or any other form of storage medium known in the art.

The previous description of the preferred embodiments is provided to enable any person skilled in the art to make or use the present invention. The various modifications to these embodiments will be readily apparent to those skilled in the art, and the generic principles defined herein may be applied to other embodiments without the use of the inventive faculty. Thus, the present invention is not intended to be limited to the embodiments shown herein but is to be accorded the widest scope consistent with the principles and novel features disclosed herein.

What we claim as our invention is:

1. An apparatus to determine a transform of a block of encoded data, the block of encoded data comprising a plurality of data elements, the apparatus comprising:

an input register configured to receive a predetermined quantity of data elements;

at least one butterfly processor coupled to the input register, the butterfly processor configured to perform at least one mathematical operation on selected pairs of data elements to produce an output of processed data elements;

at least one intermediate register coupled to the butterfly processor, the intermediate register configured to temporarily store the processed data; and

a feedback loop coupling the intermediate register and the butterfly processor, where if enabled, is configured to transfer a first portion of processed data elements to the appropriate butterfly processor to perform additional mathematical operations and, where if disabled, is configured to transfer a second portion of processed data elements to at least one holding register;

wherein the holding register is configured to store the processed data until all of the first portion data elements is processed.

2. The apparatus set forth in claim **1**, further comprising at least one input multiplexer coupling the feedback loop and the intermediate register, wherein each input multiplexer is configured to temporarily select data elements and transfer data elements to the appropriate butterfly processor.

3. The apparatus set forth in claim **1**, further comprising at least one output multiplexer coupling the butterfly processor and the intermediate register, wherein each output multiplexer is configured to temporarily select data elements and transfer data elements to the appropriate intermediate register.

4. The apparatus set forth in claim **1**, wherein the transform is selected from the group consisting of: a Discrete Cosine Transform (DCT), a Differential Quadtree Transform (DQT), an Inverse Discrete Cosine Transform (IDCT) and an Inverse Differential Quadtree Transform (IDQT).

5. The apparatus set forth in claim **1** wherein the block of encoded data may be represented as row data and column data, and further comprising a transpose random-access memory (RAM) coupled to the input register, wherein the transpose RAM is configured to store the row data while the column data is being processed, and wherein the transpose RAM is configured to store the column data while the row data is being processed.

6. The apparatus set forth in claim **5**, wherein the transpose RAM is configurable to store two blocks of encoded data.

7. The apparatus set forth in claim **5**, further comprising a write multiplexer coupling the holding register, wherein the write multiplexer is configured to resequence data elements to complete a one-dimensional transform.

8. The apparatus set forth in claim **1** wherein the feedback loop allows for the same components to be reused irrespective of block size.

9. The apparatus set forth in claim **1** wherein the feedback loop allows for the same components to be reused irrespective of the type of transform.

10. The apparatus set forth in claim **1** wherein the feedback loop allows for the same components to be reused irrespective of mathematical operation.

11. The apparatus set forth in claim **1**, further comprising a control sequencer coupled to the feedback loop, wherein the control sequencer is configured to enable or disable the feedback loop.

12. The apparatus set forth in claim **11**, where the control sequencer provides the butterfly processor with a unique coefficient multiplier.

13. The apparatus set forth in claim **12**, wherein the unique coefficient multiplier is based on B. G. Lee's algorithm.

14. The apparatus set forth in claim **11**, where the control sequencer enables certain ones of the input registers based on a predetermined event.

15. The apparatus set forth in claim **11**, where the control sequencer enables certain ones of the butterfly processors based on predetermined criteria.

16. The apparatus set forth in claim **11**, where the control sequencer enables certain ones of the intermediate registers based on predetermined criteria.

17. The apparatus set forth in claim **11**, where the control sequencer enables certain ones of the output registers based on predetermined criteria.

18. The apparatus as set forth in claim **1**, wherein the mathematical operation is from the group consisting of addition, multiplication, and subtraction.

19. The apparatus as set forth in claim **1**, wherein each butterfly processor performs a portion of a one-dimensional transform.

20. The apparatus as set forth in claim **1**, wherein the transform of a block of encoded data is computed as a series of one-dimensional transforms.

21. An apparatus to determine a transform of a block of encoded data, the block of encoded data capable of being represented as row data and column data, each row and column comprising a plurality of data elements, the apparatus comprising:

a transpose random access memory (RAM) configured to store the block of encoded data;

at least one input register coupled to the transpose RAM, the input register configured to receive columns of data from the transpose RAM;

at least one butterfly processor coupled to the input register, the butterfly processor configured to perform a portion of a one-dimensional transform on selected pairs of data elements from the column data to produce an output of first order column data;

at least one intermediate register coupled to the butterfly processor, the intermediate register configured to temporarily store the first order column data; and

a feedback loop coupling the intermediate register and the butterfly processor, where if enabled, is configured to transfer selected data elements of the first order column data to the butterfly processor to perform additional portions of one-dimensional transforms and, where if disabled, is configured to transfer the column data to the transpose RAM;

wherein the input register is then configured to receive rows of data from the transpose RAM, the butterfly processor is configured to perform a portion of a one dimensional transform on selected pairs of data elements from the rows of data to produce an output of first order row data, the intermediate register configured to temporarily store the first order row data, the feedback loop configured to transfer selected data elements of the first order row data to the butterfly processor to perform additional portions of one-dimensional transforms and, where if disabled, is configured to transfer the row data to an output register.

22. The apparatus as set forth in claim **21**, wherein the feedback loop is disabled upon completing a one-dimensional transform on the column or row data.

23. The apparatus set forth in claim **21**, further comprising at least one input multiplexer coupling the feedback loop and the intermediate register, wherein each input multiplexer is configured to temporarily select data elements and transfer data elements to the appropriate butterfly processor.

24. The apparatus set forth in claim **21**, further comprising at least one output multiplexer coupling the butterfly processor and the intermediate register, wherein each output multiplexer is configured to temporarily select data elements and transfer data elements to the appropriate intermediate register.

25. The apparatus set forth in claim **21**, wherein the transform is selected from the group consisting of: a Discrete Cosine Transform (DCT), a Differential Quadtree Transform (DQT), an Inverse Discrete Cosine Transform (IDCT) and an Inverse Differential Quadtree Transform (IDQT).

26. The apparatus set forth in claim **21**, wherein the transpose RAM is configurable to store two blocks of encoded data.

27. The apparatus set forth in claim **21**, further comprising a write multiplexer coupling the holding register, wherein the write multiplexer is configured to resequence data elements such that the one-dimensional transform is completed.

28. The apparatus set forth in claim **21** wherein the feedback loop allows for the same components to be reused irrespective of block size, type of transform or type of mathematical operation.

29. The apparatus set forth in claim **21**, further comprising a control sequencer coupled to the feedback loop, wherein the control sequencer is configured to enable or disable the feedback loop.

30. The apparatus set forth in claim **29**, where the control sequencer provides the butterfly processor with a unique coefficient multiplier.

31. The apparatus set forth in claim **29**, wherein the unique coefficient multiplier is based on B. G. Lee's algorithm.

32. The apparatus set forth in claim **29**, where the control sequencer enables certain ones of the input registers, butterfly processors, intermediate registers, or output registers based on predetermined criteria.

33. The apparatus as set forth in claim **21**, wherein the mathematical operation is from the group consisting of addition, multiplication, and subtraction.

34. The apparatus as set forth in claim **21**, wherein each butterfly processor performs a portion of a one-dimensional transform.

35. The apparatus as set forth in claim **21**, wherein the transform of a block of encoded data is computed as a series of one-dimensional transforms.

36. An apparatus to perform an N dimensional transform as a cascade of N one-dimensional transforms on a block of encoded data, the encoded data comprising a plurality of data elements, the apparatus comprising:

a cluster of butterfly processors coupled to the input register, each butterfly processor configured to perform a portion of a one-dimensional transform on selected pairs of data elements to produce an output of partially processed data comprising a plurality of partially processed data elements;

at least one intermediate register coupled to each butterfly processor, the intermediate register configured to temporarily store the partially processed data; and

a feedback loop coupled to the intermediate register and the butterfly processor, where the feedback loop is enabled as necessary to route selected pairs of the partially processed data elements to the appropriate butterfly processor to perform additional portions of one-dimensional transforms until a one dimensional transform is completed.

37. The apparatus set forth in claim **36**, wherein the transform is selected from the group consisting of: a Discrete Cosine Transform (DCT), a Differential Quadtree Transform (DQT), an Inverse Discrete Cosine Transform (IDCT) and an Inverse Differential Quadtree Transform (IDQT).

38. The apparatus set forth in claim **36** wherein the block of encoded data may be represented as row data and column data, and further comprising a transpose read-only memory (RAM) coupled to the input register, wherein the transpose RAM is configured to store the row data while the column data is being processed, and wherein the transpose RAM is configured to store the column data while the row data is being processed.

39. The apparatus set forth in claim **38**, wherein the transpose RAM is configurable to store two blocks of encoded data.

40. The apparatus set forth in claim **36** wherein the feedback loop allows for the same components to be reused irrespective of block size, type of transform or type of mathematical operation.

41. The apparatus set forth in claim **36**, further comprising a control sequencer coupled to the feedback loop, wherein the control sequencer is configured to enable or disable the feedback loop.

42. The apparatus set forth in claim **41**, where the control sequencer provides the butterfly processor with a unique coefficient multiplier.

43. The apparatus set forth in claim **42**, wherein the unique coefficient multiplier is based on B. G. Lee's algorithm.

44. The apparatus set forth in claim **41**, where the control sequencer enables certain ones of the input registers, butterfly processors, intermediate registers, or output registers based on predetermined criteria.

45. An apparatus to determine the inverse discrete cosine transform of a block of encoded data, the block of encoded data comprising a plurality of data elements, the apparatus comprising:

an input register configured to receive a predetermined quantity of data elements;

at least one butterfly processor coupled to the input register, the butterfly processor configured to perform at least one mathematical operation on selected pairs of data elements to produce an output of processed data elements;

at least one intermediate register coupled to the butterfly processor, the intermediate register configured to temporarily store the processed data; and

a feedback loop coupling the intermediate register and the butterfly processor, where if enabled, is configured to transfer a first portion of processed data elements to the appropriate butterfly processor to perform additional mathematical operations and, where if disabled, is configured to transfer a second portion of processed data elements to at least one holding register;

wherein the holding register is configured to store the processed data until all of the first portion data elements is processed.

46. An apparatus to determine a transform of a block of encoded data, the block of encoded data capable of being represented as row data and column data, each row and column comprising a plurality of data elements, the apparatus comprising:

a transpose random-access memory (RAM) configured to store the block of encoded data;

at least one input register coupled to the transpose RAM, the input register configured to receive columns of data from the transpose RAM;

at least one butterfly processor coupled to the input register, the butterfly processor configured to perform a first order transform on selected pairs of data elements from the column data to produce an output of first order column data;

at least one intermediate register coupled to the butterfly processor, the intermediate register configured to temporarily store the first order column data;

a feedback loop coupling the intermediate register and the butterfly processor, where if enabled, is configured to transfer selected data elements of the first order column data to the butterfly processor to perform additional transforms and, where if disabled, is configured to transfer the column data to the transpose RAM; and

a control sequencer coupled to the feedback loop, wherein the control sequencer is configured to enable or disable the feedback loop

wherein the input register is then configured to receive rows of data from the transpose RAM, the butterfly processor is configured to perform a first order transform on selected pairs of data elements from the rows of data to produce an output of first order row data, the intermediate register is configured to temporarily store the first order row data, the feedback loop is configured to transfer selected data elements of the first order row data to the butterfly processor to perform additional transforms and, where if disabled, is configured to transfer the row data to an output register.

47. A method to determine a transform of a block of encoded data, the block of encoded data comprising a plurality of data elements, the method comprising:

(a) receiving a predetermined quantity of data elements;

(b) performing at least one mathematical operation on selected pairs of data elements to produce an output of processed data elements;

(c) making a determination as to whether any of the processed data elements require additional mathematical operations;

(d) selecting a first portion of processed data elements that require additional mathematical operations;

(e) selecting a second portion of processed data elements that do not require additional mathematical operations;

(f) performing at least one mathematical operation on selected pairs of the first portion of processed data elements to produce a second output of processed data elements; and

(g) storing the second portion of processed data elements until all of the first portion of data elements is processed.

48. The method set forth in claim **47**, further comprising:

(h) repeating steps (c), (d), (e), (f) and (g) as necessary.

49. The method set forth in claim **47**, further comprising:

(i) outputting the block of encoded data when all of the data elements of the block of encoded data do not require additional mathematical operations.

50. The method set forth in claim **47**, wherein the transform is selected from the group consisting of: a Discrete Cosine Transform (DCT), a Differential Quadtree Transform (DQT), an Inverse Discrete Cosine Transform (IDCT) and an Inverse Differential Quadtree Transform (IDQT).

51. The method set forth in claim **47** wherein the block of encoded data may be represented as row data and column data, and further comprising:

storing the row data while the column data is being processed; and

storing the column data while the row data is being processed.

52. The method set forth in claim **47**, further comprising resequencing data elements before the step of storing, such that subsequent delivery of data elements is performed in an efficient manner.

53. The method set forth in claim **47**, further comprising controlling steps (a), (b), (c), (d), (e), (f), (g), and (h) based upon predetermined criteria.

54. The method set forth in claim **53**, further comprising providing a unique coefficient multiplier to certain data elements based upon predetermined criteria.

55. The apparatus set forth in claim **54**, wherein the unique coefficient multiplier is based on B. G. Lee's algorithm.

56. The method set forth in claim **47**, wherein the mathematical operation is from the group consisting of addition, multiplication, and subtraction.

57. The method as set forth in claim **47**, wherein each butterfly processor performs a portion of a one-dimensional transform.

58. The method as set forth in claim **47**, wherein the transform of a block of encoded data is computed as a series of one-dimensional transforms.

59. A computer readable medium containing constructions for controlling a computer system to perform a method, the method comprising:

(a) receiving a predetermined quantity of data elements;

(b) performing at least one mathematical operation on selected pairs of data elements to produce an output of processed data elements;

(c) making a determination as to whether any of the processed data elements require additional mathematical operations;

(d) selecting a first portion of processed data elements that require additional mathematical operations;

(e) selecting a second portion of processed data elements that do not require additional mathematical operations;

(f) performing at least one mathematical operation on selected pairs of the first portion of processed data elements to produce a second output of processed data elements; and

(g) storing the second portion of processed data elements until all of the first portion of data elements is processed.

60. An apparatus to determine a transform of a block of encoded data, the block of encoded data comprising a plurality of data elements, the apparatus comprising:

(a) means for receiving a predetermined quantity of data elements;

(b) means for performing at least one mathematical operation on selected pairs of data elements to produce an output of processed data elements;

(c) means for making a determination as to whether any of the processed data elements require additional mathematical operations;

(d) means for selecting a first portion of processed data elements that require additional mathematical operations;

(e) means for selecting a second portion of processed data elements that do not require additional mathematical operations;

(f) means for performing at least one mathematical operation on selected pairs of the first portion of processed data elements to produce a second output of processed data elements; and

(g) means for storing the second portion of processed data elements until all of the first portion of data elements is processed.

61. The apparatus set forth in claim **47**, further comprising:

(h) means for repeating steps (c), (d), (e), (f) and (g) as necessary.

62. The apparatus set forth in claim **47**, further comprising:

(i) means for outputting the block of encoded data when all of the data elements of the block of encoded data do not require additional mathematical operations.

63. The apparatus set forth in claim **47**, wherein the transform is selected from the group consisting of: a Discrete Cosine Transform (DCT), a Differential Quadtree Transform (DQT), an Inverse Discrete Cosine Transform (IDCT) and an Inverse Differential Quadtree Transform (IDQT).

64. The apparatus set forth in claim **47** wherein the block of encoded data may be represented as row data and column data, and further comprising:

means for storing the row data while the column data is being processed; and

means for storing the column data while the row data is being processed.

65. The apparatus set forth in claim **47**, further comprising means for resequencing data elements before the step of storing, such that subsequent delivery of data elements is performed in an efficient manner.

66. The apparatus set forth in claim **47**, further comprising means for controlling elements (a), (b), (c), (d), (e), (f), (g), and (h) based upon predetermined criteria.

67. The apparatus set forth in claim **66**, further comprising providing a unique coefficient multiplier to certain data elements based upon predetermined criteria.

68. The apparatus set forth in claim 67, wherein the unique coefficient multiplier is based on B. G. Lee's algorithm.

69. The apparatus set forth in claim 60, wherein the mathematical operation is from the group consisting of addition, multiplication, and subtraction.

70. The apparatus as set forth in claim 60, wherein each butterfly processor performs a portion of a one-dimensional transform.

71. An apparatus to determine a transform of encoded data, the encoded data comprising a plurality of data elements in the pixel domain, the apparatus comprising:

a block size assigner configured to receive the plurality of data elements and group the elements into a plurality of groups of data elements in the pixel domain;

a DCT/DQT transformer configured to transform the data elements from the pixel domain to the frequency domain, the transformer further comprising:

an input register configured to receive a predetermined quantity of data elements of the group;

at least one butterfly processor coupled to the input register, the butterfly processor configured to perform at least one mathematical operation on selected pairs of data elements to produce an output of processed data elements;

at least one intermediate register coupled to the butterfly processor, the intermediate register configured to temporarily store the processed data; and

a feedback loop coupling the intermediate register and the butterfly processor, where if enabled, is configured to transfer a first portion of processed data elements to the appropriate butterfly processor to perform additional mathematical operations and, where if disabled, is configured to transfer a second portion of processed data elements to at least one holding register;

wherein the holding register is configured to store the processed data until all of the first portion data elements is processed;

a quantizer configured to quantize the frequency domain elements to emphasize those elements that are more sensitive to the human visual system, and de-emphasize those elements that are less sensitive to the human visual system;

a serializer configured to produce a serialized stream of frequency domain elements; and

a variable length coder configured to determine successive frequency domain elements and non-successive frequency domain elements.

72. The apparatus set forth in claim 71, further comprising at least one input multiplexer coupling the feedback loop and the intermediate register, wherein each input multiplexer is configured to temporarily select data elements and transfer data elements to the appropriate butterfly processor.

73. The apparatus set forth in claim 71, further comprising at least one output multiplexer coupling the butterfly processor and the intermediate register, wherein each output multiplexer is configured to temporarily select data elements and transfer data elements to the appropriate intermediate register.

74. The apparatus set forth in claim 71 wherein the block of encoded data may be represented as row data and column data, and further comprising a transpose random-access memory (RAM) coupled to the input register, wherein the transpose RAM is configured to store the row data while the column data is being processed, and wherein the transpose RAM is configured to store the column data while the row data is being processed.

75. The apparatus set forth in claim 74, wherein the transpose RAM is configurable to store two blocks of encoded data.

76. The apparatus set forth in claim 74, further comprising a write multiplexer coupling the holding register, wherein the write multiplexer is configured to resequence data elements to complete a one-dimensional transform.

77. The apparatus set forth in claim 71 wherein the feedback loop allows for the same components to be reused irrespective of block size.

78. The apparatus set forth in claim 71, further comprising a control sequencer coupled to the feedback loop, wherein the control sequencer is configured to enable or disable the feedback loop.

79. The apparatus set forth in claim 78, where the control sequencer provides the butterfly processor with a unique coefficient multiplier.

80. The apparatus set forth in claim 78, where the control sequencer enables certain ones of the input registers based on a predetermined event.

81. The apparatus set forth in claim 78, where the control sequencer enables certain ones of the butterfly processors based on predetermined criteria.

82. The apparatus set forth in claim 78, where the control sequencer enables certain ones of the intermediate registers based on predetermined criteria.

83. The apparatus set forth in claim 78, where the control sequencer enables certain ones of the output registers based on predetermined criteria.

84. The apparatus as set forth in claim 71, wherein the mathematical operation is from the group consisting of addition, multiplication, and subtraction.

85. The apparatus as set forth in claim 71, wherein each butterfly processor performs a portion of a one-dimensional transform.

86. A method of transforming encoded data from the pixel domain to the frequency domain, the encoded data comprising a plurality of data elements, the method comprising:

(a) grouping the plurality of data elements in the pixel domain into a plurality of blocks, each block comprising a plurality of data elements in the pixel domain;

(b) performing at least one mathematical operation on selected pairs of data elements to produce an output of processed data elements;

(c) making a determination as to whether any of the processed data elements require additional mathematical operations;

(d) selecting a first portion of processed data elements that require additional mathematical operations;

(e) selecting a second portion of processed data elements that do not require additional mathematical operations;

(f) performing at least one mathematical operation on selected pairs of the first portion of processed data elements to produce a second output of processed data elements;

(g) storing the second portion of processed data elements until all of the first portion of data elements is processed;

(h) repeating steps (c), (d), (e), (f) and (g), as necessary, until all of the data elements do not require additional mathematical operations and are converted to frequency domain elements;

(i) quantizing the frequency domain data elements to emphasize those elements that are more sensitive to the human visual system and de-emphasize those elements that are less sensitive to the human visual system;

(j) serializing the quantized frequency domain data elements to produce a serialized stream of frequency domain elements; and

(k) coding the serialized frequency domain elements to determine successive frequency domain elements and non-successive frequency domain elements.

87. The method set forth in claim **86** wherein the block of encoded data may be represented as row data and column data, and further comprising:

storing the row data while the column data is being processed; and

storing the column data while the row data is being processed.

88. The method set forth in claim **86**, further comprising controlling steps (a), (b), (c), (d), (e), (f), (g), and (h) based upon required control signals.

89. The method set forth in claim **88**, further comprising providing a unique coefficient multiplier to certain data elements based upon predetermined criteria.

90. The method as set forth in claim **86**, wherein each butterfly processor performs a portion of a one-dimensional transform.

91. The method as set forth in claim **86**, wherein the transform of a block of encoded data is computed as a series of one-dimensional transforms.

* * * * *